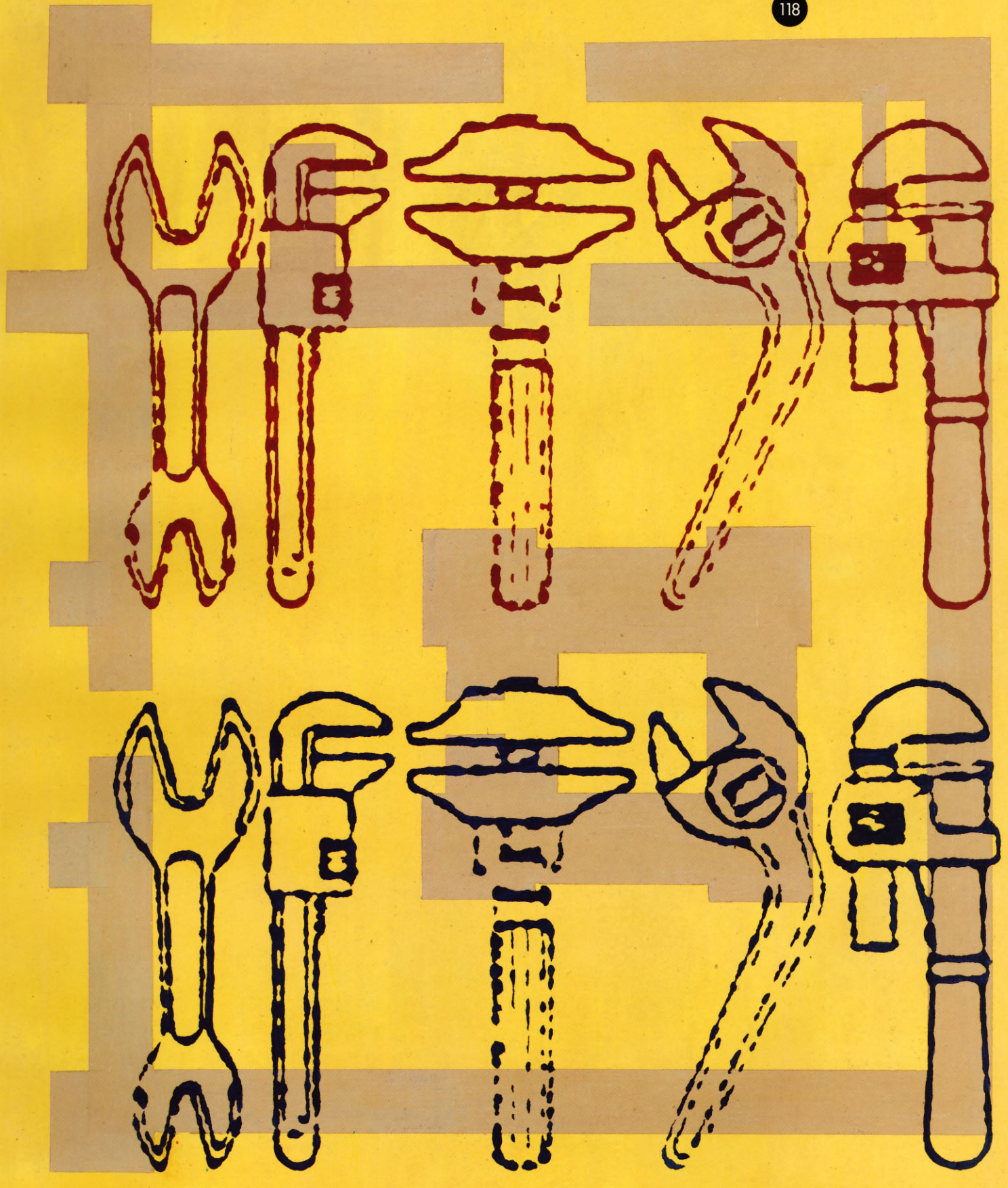
novatica

Revista
de la Asociación
de Técnicos
de Informática

NOVIEMBRE-DICIEMBRE 1995

118



INGENIERÍA DEL SOFTWARE

novática

Revista de la Asociación de Técnicos de Informática

Novática es el órgano oficial y de formación contínua de la Asociación de Técnicos de Informática (ATI)

ATI tiene un acuerdo internacional con ACM y como miembro de FESI, colabora con IFIP y CEPIS. ADA-Spain está vinculada a ATI.

Coordinación y Redacción (ATI Valencia)
Palomino 14, 2ª, 46003 Valencia
(96)3918531; fax 3918531

Administración y Publicidad (ATI Cataluña)
Via Laietana 41, 1º, 1ª, 08003 Barcelona
(93)4125235; fax 4127713 / secregen@ati.es

Delegación (ATI Madrid)
Padilla 66, 3º, 28006 Madrid;
(91)4029391; fx.3093685/secremdr@atimdr.es

Delegación (ATI Andalucía)
Avda. República Argentina 25, 4ª 41011 Sevilla
(95) 4278198; fax 4274623 (provisionales)

Imprenta: CIMAGOTIPO, S.L.
Pallars, 161, 08005 Barcelona
Depósito Legal: B 15.154-1975
ISBN: 0211-2124; CODEN NOVAEC

Sumario: noviembre y diciembre de 1995

Monografía: Ingeniería del software

Secciones técnicas

Sociedad de la Información

Ingeniería del Software

Presentación

Juan Carlos Granja
Profesor Responsable del Grupo de Investigacion de Lenguajes y Sistemas Informáticos e Ingeniería del Software de la Universidad de Granada

¿'Pastillas' de software...? La revista norte US NEWS & WORL REPORT, ha publicado recientemente un amplio estudio, donde identifica los mejores trabajos en el futuro. Aparece en lugar muy destacado INGENIERO DEL SOFTWARE.

A pesar de las reestructuraciones laborales vividas en el sector informático, no son una sorpresa los resultados del estudio. El desarrollo de software tiene futuro según este informe, en cuanto se acomoda a los nuevos retos. Los ingenieros que sacian la sed de software se han convertido en un bien muy preciado. Sus ventas en EEUU han crecido una media del 27 por 100 anual desde 1982. El año pasado se ingresaron 35.600 millones de dólares por software comercial y se prevé que esa cantidad se duplique con creces de aquí al año 2000.

A medida que los computadores y las redes se hacen más complejos, ocurre lo mismo con las expectativas de los consumidores; por ello, el desarrollo de software (como verdadera solución informática frente a otras) debe modernizarse continuamente para ser mejor, más eficiente y conseguir unos niveles de calidad que permitan el resultado que el usuario final demanda. La Universidad Carnegie Mellon afirma que la contratación de sus estudiantes de ingeniería del software ha crecido más de un 20 por 100 este año.

Toda la continua modernización de hoy día en ingeniería del software y la posibilidad de dar respuesta a la mayoría de las necesidades de desarrollo de software contrastan con la realidad de un pasado próximo. Cuando asumí la coordinación de esta monografía sobre ingeniería del software, me vino el recuerdo de los que en los años setenta y principios de los ochenta afrontábamos proyectos de desarrollo de software en la industria o la administración. Ante los escasos medios de desarrollo de que se disponía, frecuentemente aparecía la referencia a autores que pugnaban por realizar en el campo del software lo que ya se conseguía en otros ámbitos. Los componentes **CI (circuitos integrados)** tienen un número de pieza, una función determinada, una interfaz bien definida y un conjunto estándar de criterios de integración. En definitiva actuan como verdaderas 'pastillas'.

"¿Porqué no hay 'pastillas de software' ?" volvemos a oír, planteada por nuestros alumnos, 'la' pregunta de entonces; ¿ cúando será posible que funcione con efectividad la reusabilidad del fruto de nuestro trabajo, conseguido con tanto tiempo y desvelo...? Evidentemente, el uso cada vez más extendido de la programación orientada a los objetos ha dado como resultado la creación de auténticas pastillas de software. Lejos queda ya el hecho histórico de la ciudad de Garmisch, donde en 1968 se acuñó el término de ingeniería del software.

"Debido a los cambios rápidos e inexorables que se están produciendo en la tecnología de la información y a las consecuencias irreversibles de quedarse atrás, las empresas se ven obligadas a asimilar la tecnología o morir...Esta situación se parece a una rueda de molino tecnológica. Las empresas tendrán que trabajar cada vez más para estar actualizadas". Esto decía Max Hopper en 1990 en su trabajo titulado *Rattling SABRE, News Ways to Compete on Information*, de la Harvard Business Review (mayo-junio)..

Como hemos podido comprobar toda nueva aportación sufre un proceso, y como afirma Michael Horner de Digital Equipment Corporation, todo nuevo concepto suele requerir unos 15 años para pasar de idea inicial a producto de masas: de forma que en los 5 primeros años se formula una idea nueva que se plasma en un prototipo utilizado para demostrar los conceptos básicos; en los 5 años siguientes los científicos proceden a refinar el prototipo y también se procede a lanzar los primeros productos; y por último se introduce el producto en el mercado comercialmente en los 5 años siguientes, completando así el 'ciclo' de 15 años expresado por Horner. Aplicando este ciclo completo desde la simple idea a un producto de masas, recordemos que las tecnologías orientadas a los objetos se iniciaron en los años ochenta... ¿podríamos decir que estamos en el último tramo de la regla del ciclo de 15 años?

Como es de todos conocido, se celebran dentro del campo de la Ingeniería del Software, en sus diferentes vertientes, importantes eventos con distinta periodicidad y lamentablemente no con la frecuencia y abundancia que los interesados en estos temas quisiéramos. Dada la historia que arrastra y por lo acontecido en él, merece especial mención el **ESEC'95**. Se desarrolló en Sitges del 25 al 28 de septiembre. En ella ATI actuaba como anfitriona y su contribución se ha realizado dentro del marco de CEPIS, el Comité Europeo de Sociedades de Profesionales de Informática. ATI ha sostenido a la organización del Congreso Europeo, buscando siempre la divulgación de las Tecnologías de la Información, en este caso en el campo de la ingeniería del software.

Tenemos la enorme satisfacción de iniciar este número con una aportación de primera mano, la del *Executive Chairman* del ESEC'95, Profesor Pere Botella, Decano de la Facultad de Informática de Barcelona. Además de sus apreciadas opiniones, su aportación refleja la encuesta realizada a los asistentes por la organización del ESEC'95. ATI mostró su apoyo a la Conferencia y el siguiente apartado de la monografía recoge las palabras del Presidente de ATI pronunciadas en la inauguración de ESEC'95. No puede extrañar que ATI siga con su tradicional y aquí especial apoyo a los acontecimientos que permiten la formación y actualización de conocimientos de los profesionales de la Informática.

Para ofrecer una opinión del desarrollo de ESEC'95, cabe destacar desde el primer día una notable asistencia a los tutoriales. Rubén Prieto-Diaz de Reuse Inc. nos deleitó con su análisis del dominio de la reusabilidad, dando una vertiente eminentemente práctica a su exposición. Fue muy animado el coloquio final que permitió ampliar determinados aspectos de interés. Hans-Martin Hoercher nos dió su particular visión del papel de la especificación formal en el testeo del software. Interesante fue la presentación del tutorial de Philippe Kruchten de la Rational Western Canada, que hizo un amplio recorrido por la problemática de los procesos iterativos de desarrollo y de la arquitectura del software. Fue seguido con gran interés el desarrollo del tutorial de Mehdi Jazayeri de la Technische Universitae de Viena que hizo una exposición de la problemática del diseño del software y la implementación en C++. Richard Kemmerer de la Universidad de California sorprendió con su ilustración de los aspectos de la seguridad en los sistemas

informáticos a los pocos asistentes, que tuvieron la oportunidad de hacer un recorrido ameno por la seguridad en redes.

Para entrar de lleno en el contenido de los debates, no puede evitarse la tradicional pregunta ¿estamos en una etapa de transición en la ingeniería del software? En esa etapa de 'transición' en que, en opinión de algunos, se encuentra la ingeniería del software, creemos que ESEC'95 ha marcado al menos nuevas pautas con las conferencias invitadas. La primera sobre la visión de la demanda industrial de ingenieros de software no revistió especial relevancia aclaratoria. Pero en las siguientes conferencias volvieron a aparecer las 'pastillas de software'. Hay que destacar el enfrentamiento entre Bertran Meyer y François Bancilhon con su distinta visión de la justificación a la necesidad de utilización de bases de datos orientadas a objetos. Fue uno de los momentos estrella del Congreso, junto con la conferencia de Watts Humphrey, que nos deleitó a todos con su visión de la calidad del software, y que puso un merecido broche de oro, tras el animado coloquio que resaltó el interés de los asistentes por la calidad del sofware (las encuestas reflejaron la máxima puntuación). Más controversia hubo en el panel sobre plataformas abiertas y distribuidas, que fue de interés para muchos por los distintos aspectos afrontados.

En las diez sesiones del Congreso, las 29 ponencias presentadas y retenidas cubrieron con un nivel de calidad alto las distintas parcelas de la ingeniería del software. Los asistententes siguieron con gran interés las sesiones, dando lugar a animados coloquios al final de cada exposición. Sería muy difícil destacar una sesión. Interés para los asistentes tuvo la sesión del *Technical Council on Software Engineering* de IEEE, las exposiciones de productos y el *Industrial Track*. Especial asistencia y seguimiento tuvo el Workshop del proyecto **Nature**, pese a la cantidad de sesiones y ponentes. Fue muy participativo el coloquio final de cada sesión y hay que destacar la abundancia de documentación presentada por los organizadores (Nature es un proyecto ESPRIT centrado en **ingeniería de requisitos** pero que afronta gran cantidad de vertientes recogidas en el alto número de informes presentados). Se prodría hablar de sobresaliente el interés despertado.

ESEC'95 también tuvo sus detalles de organización. Nota de interés lógico para muchos fue la posibilidad de usar Internet en una de las salas del Congreso, muy valorada y agradecida por los asistentes. La climatología contraria no fue un obstáculo para la gran capacidad de los organizadores que supieron afrontar perfectamente los retos que se presentaban. El grupo Ars-Animae nos dejó gratamente impresionados a muchos.

Volviendo a la organización concreta de esta monografía y después de planear distintas estructuraciones para ordenar los artículos, se pensó que todas ellas darían lugar a polémica ante las distintas visiones en un sector tan polifacético. Así que se ha optado por una alternancia entre las aportaciones del ámbito de la industria y las del universitario. El resultado intenta reflejar, tanto por la disposición como por el número de aportaciones, el profundo equilibrio entre los dos ámbitos que juntos han hecho y continuarán haciendo mucho por la ingeniería del software. Por poner otro ejemplo reciente, en las Jornadas Técnicas centradas en ingeniería del software y organizadas por ATI en la E.T.S.de Ingeniería Informática de Granada durante octubre y noviembre, también se puso de manifiesto el interés mostrado por asistentes de la industria, la administración, de profesores y de alumnos, transformadas en altas a ATI procedentes de todos los colectivos citados.

Repasando el contenido de artículos de la monografía, tras la crónica de ESEC'95 por Pere Botella y el saludo a la Conferencia de pedro Gómez Grau, la aportación de Jose Luis Iparraguirre, del European Software Institute, expone los métodos de evaluación y mejora de los procesos de producción del software. Una autoridad en este campo, un grupo de autores que incluye al Profesor Isidro Ramos Salavert presenta un importante trabajo sobre una arquitectura orientada a agentes utilizando un nuevo lenguaje como soporte lingüístico que ofrece la visión del sistema de información como una colección de agentes. Juan Peña y Jesus Macías aportan una experiencia práctica de utilización y uso metodológico. Ambrosio Toval y otros autores realizan un prototipado, ejecución de especificaciones basadas en diagramas de flujo de datos. Josep Ramón Freixanet y otros autores presentan su trabajo de gran interés para obtener una herramienta parametrizable. Alberto Valderruten, Nuria Castell y Javier Tuya fueron ponentes en ESEC'95 y aportan un importante trabajo de su linea de investigación. Jose Luis Esteban y Mario Piattini, dos compañeros de AENOR 71/SC7, transmiten una aportación en normalización internacional. Es de destacar la experiencia práctica en 'calidad en el mantenimiento de un sistema' que presentan I. García y J.A. Mellado. Dentro del ámbito de calidad del software, se presenta un original estudio en Eurométodo. Fernando Aldea presenta aportaciones sobre software en el sector espacial. Dentro de las posibilidades emergentes nos encontramos con el trabajo del entorno DELPHOS. La aportación de las técnicas de la información nos da una visión bastante interesante de la situación en España y las perspectivas futuras. De las posibilidades de utización de las técnicas formales y su impacto en la calidad, encontramos un interesante trabajo de G. Huecas. Tras dos artículos sobre Reutilización de Código e Inspecciones de calidad escritos por F. Aldea y otros autores, cierra la monografía un último artículo global sobre análisis y perspectivas de A. Bajja.

Es muy difícil indicar tendencias futuras en un campo cada vez más dinámico que evoluciona por el influjo de avances tecnológicos internos y la presión de un entorno que exige continuas soluciones a sus nuevas necesidades. Parece confirmarse una preocupación creciente por la calidad de un software que satisfaga al usuario y permita la eficiencia en las prestaciones y la facilidad en el mantenimiento.

Las tecnologías orientadas a los objetos podrían formar un puente entre los enfoques de inteligencia artificial (inherentemente orientada a los objetos), las aplicaciones convencionales y la tecnología de las bases de datos. El diseño orientado a los objetos proporciona un método para romper las barreras entre los datos y el procesamiento, con la consiguiente mejora de la calidad del software. A medida que los enfoques orientados a los objetos tomen más relevancia, los paradigmas evolutivos para la ingeniería del software se irán modificando para integrar la reusabilidad de componentes de programas. La reusabilidad es una característica importante para un componente de software de alta calidad. La calidad, como campo de investigación, nunca perderá su importancia.

Después de todo lo expuesto y mirando al horizonte del siglo veintiuno, sería interesante prestar atención retrospectiva a Rich y Waterls. En su libro *The Programmer's Apprentice* (Addison-Wesley, 1990) ya hacía de una forma bastante razonable el recorrido de lo que ya ocurre actualmente y lo que puede suceder en el futuro próximo respecto al desarrollo del software y los proyectos informáticos ¿será posible la 'pastilla de software' llevada a sus últimas consecuencias...?

Ingeniería del Software

Crónica de ESEC´95 (European Software Engineering Conference 1995)

Pere Botella
ESEC'95 Executive Chair

A petición de ***Novática*** escribo estas breves notas sobre el desarrollo de la conferencia internacional ESEC'95, que tuvo lugar en Sitges (Barcelona) del 25 al 28 de Setiembre pasado, y en la que ATI actuaba como sociedad anfitriona. Atiendo esta petición con gusto, pero insistiendo en lo que ya he manifestado a los responsables de la revista: por ser el responsable de la conferencia (Executive Chair, en la jerga habitual), me pasé esos días cómo loco, de un lado a otro, decidiendo sobre la marcha y tratando de apañar los problemas inevitables que suelen surgir, así que no estoy muy seguro de haberme enterado de lo que ocurría en las salas, a las que casi no entré. En todo caso, me referiré a las opiniones de los asistentes, ya que disponemos de una encuesta.

Antecedentes

La serie de conferencias ESEC fue fundada por cuatro sociedades de profesionales informáticos: AICA (Italia), AFCET (Francia), BCS (Reino Unido) y GI (Alemania). Si bien no era la primera conferencia internacional (informática) de carácter europeo, sí era la primera co-fundada por sociedades de profesionales y con carácter europeísta. El porqué de ese tema (la Ingeniería del Software) y no otro, fué circunstancial: todas las sociedades implicadas tenían grupos internos en el tema y experiencia en organizar conferencias. El fruto de la cooperación fué ESEC'87, celebrado en Estrasburgo. Poco después, fuí invitado a formar parte del comité permanente (steering comittee) de ESEC: además de trabajar en Ingeniería del Software, en el comité había y hay personas con las que mantengo una fuerte amistad. Aprovechando la voluntad del comité de ampliar el grupo, ATI entró, a la vez que SI (Suiza) y naturalmente con la autorización de la Junta de ATI de entonces.

La serie continuó con ESEC'89 (en Warwick), ESEC'91 (en Milano) y ESEC'93 (en Garmisch, celebrando el 25 aniversario de la invención, en ese mismo lugar, del término 'ingeniería del software'), todas ellas co-patrocinadas por AICA, AFCET, ATI, BCS, GI y SI. Tras Garmisch, las cuatro sociedades fundadoras habían organizado ESEC en su país. Los siguientes, pues, eramos los suizos y nosotros: los suizos optaron por el 97, así que nosotros presentamos la candidatura para el 95. Pero arrancamos en una situación difícil, ya que Garmisch, a pesar de un excelente programa y de un excelente lugar, generó un fuerte déficit. Ante esa situación, CEPIS (Council of European Professional Informatics Societies), en la que se agrupan las seis sociedades citadas junto a las del resto de Europa, firmó un acuerdo con ESEC para asumir la conferencia como promotor único, incluyendo el riesgo económico. Pero el acuerdo se firmó con ESEC'95 en marcha, de modo que para esta edición se hizo un protocolo 'ad-hoc': ATI asumió el 40% del riesgo, quedando el 60% restante repartido entre varios miembros de CEPIS. Así llegamos a Sitges...

Las cifras

ESEC'95 fué bien, en todos los sentidos, incluído el económico. No sólo no se ha perdido dinero, sino que va a quedar un cierto beneficio (en el momento de redactar estas líneas aún no hemos podido cerrar cuentas), lo cual, para los tiempos que corren, es un éxito. ESEC recibió un total de 170 asistentes de 22 países. Por una vez, los locales quedamos bien ya que nuestro país aportó 36 congresistas, el que más (y digo 'por una vez', ya que a pesar del aparente interés por la ingeniería del software que parece detectarse por aquí, en los ESEC precedentes o en otros even-tos del mismo tema no superamos la cifra de ... entre uno y tres asistentes españoles). Alemania, con 35, fue el otro gran con-tribuyente, seguido por 17 de USA, 12 de Reino Unido y Suecia, 10 de Italia, 6 de Austria, Francia y Holanda, 5 de Suiza, 4 de Finlandia y Noruega, 2 de Bélgica y un asistente de Australia, Canadá, Eslovenia, Irlanda, Nueva Zelanda, Portugal, Singapur y Trinidad (atención: la lista de países que tengo es de 26/09, a medio congreso, faltando inscritos de última hora).

Actos previos: Tutoriales, Workshop y Recepción

El lunes 25 estuvo ocupado por los tutoriales previstos organizados por Gregor Engels (tutorial chair) que tuvieron un total de 45 inscritos a 5 temas: Análisis de dominios para reusabilidad, impartido por Rubén Prieto-Díaz (Reuse Inc.); Arquitecturas Software y Proceso de desarrollo iterativo, por Philippe Kruchten (Rational); Diseño e implementación de software con componentes C++, por Mehdi Jazayeri y Georg Trausmuth (Univ. Tecn. de Viena); Introducción a la Seguridad Informática, por Richard Kemmerer (Univ. de California); y el rol de las especificaciones formales en el test de software, por Hans-Martin Hoercher (DST). Por comentarios oídos y por las encuestas, un alto grado de satisfacción. Chocante, por no usar otro término, que un tutorial sobre seguridad (virus) muy orientado a redes (piensen en Internet y en la WWW), impartido por una autoridad mundial sobre el tema, reúna a cuatro gatos: eso sí, se lo pasaron en grande.

En paralelo organizamos un Workshop del proyecto NATURE, un proyecto ESPRIT sobre Ingeniería de requisitos. Estaban los del proyecto, pero era abierto a todos los congresistas. Sala a rebosar (unas 40 personas) y caras de satisfacción a la salida.

La recepción de bienvenida estaba prevista en las terrazas del Palau Maricel, por gentileza del Ayuntamiento de Sitges, pero una tormenta de granizo impresionante (para los de fuera, los mediterraneos ya estamos familiarizados con las gotas frías de esa época) forzó un cambio de escenario, y nos quedamos en el hotel, tras montar una red improvisada para avisar a los despistados. Aunque el marco no era el deseado, todo fué bien: bienvenida a cargo de representantes del Ayuntamiento y de la organización, un delicioso concierto coral por Ars-Animae con canciones renacentistas españolas y populares catalanas, y un surtido aperitivo servido por el hotel Antemare.

La conferencia

El acto de apertura tuvo unas palabras de Giulio Occhini y y de Pedro Gómez-Grau, presidentes de CEPIS y de ATI, tras las

que Wilhelm Schäfer (program chair) y yo mismo comentamos temas del programa y de los actos paralelos y sociales.

De las cuatro conferencias invitadas, la primera sobre la visión industrial de la demanda profesional de ingenieros de software, impartida por Heinz.G.Schwärtzel (FAST) fué la que menos gustó a tenor de las encuestas, aunque superó el aprobado (2,6 sobre 5). El debate, en forma de dos conferencias consecutivas, entre François Bancilhon (O2 technology) y Bertrand Meyer (ISE) sobre Bases de Datos orientadas a objetos, generó expectación y no defraudó (3,8 y 3,5 sobre 5). Watts Humphrey, en la clausura, hablándonos de calidad y de su 'Personal Software Process', se llevó los mejores aplausos (4,5 sobre 5). Un sólo panel (o mesa redonda). El tema: tendencias en plataformas abiertas y distribuídas, moderado y organizado por Gonzalo León (DIT-UPM). No pude asistir (como era de esperar) pero detecté división de opiniones: entusiasmo de algunos y muecas de disgusto en otros. La evaluación de la encuesta da el promedio: 3,1 sobre 5.

De los artículos, 29 repartidos en 10 sesiones, puedo hablar de un nivel muy alto en calidad por las actas (publicadas por Springer-Verlag) y de un impecable funcionamiento de las sesiones (gracias a los moderadores). De las presentaciones, puedo dar la nota media de la encuesta: 3,3 sobre 5. En general, los comentarios eran de satisfacción.

Actos complementarios

En el hall de la salas de conferencia se habilitaron unos stands. Estuvieron presentes: European Software Institute, MTR y asociados (System Architect), NATURE (proyecto Esprit), PROTEUS (proyecto ESPRIT), VERILOG (Logiscope y LOV/OMT) y las editoriales Prentice-Hall y John-Wiley: nota: 2,9. En una sala aneja y en paralelo con las sesiones, los expositores hacían sus presentaciones, bajo el nombre de 'Industrial Track': nota, 3,2. Si bien ambas partes mantuvieron un nivel alto de calidad, era floja en cantidad, aunque la organización hizo lo que pudo y más para conseguir expositores. Hubo tambien una reunión del TCSE (Technical Council on Software Engineering) de IEEE, animada por Elliott Chikofsky ('chair' del TCSE, miembro de la Junta de la IEEE Computer Society y seguramente recordado por los asistentes a CIL'91, donde habló de CASE), con una participación muy activa de los miembros europeos de TCSE.

La cena del congreso (Catalan evening) se hizo en el Palau Novella, en pleno parque natural del Garraf, en un lugar bonito aunque de acceso algo difícil. Los problemas de los microbuses en las pronunciadas rampas pusieron nervioso a más de uno. Pero se les pasó a la vista de la abundancia de 'fuet' y aprendiendo a beber vino del porrón. La actuación de los 'Castellers de Gavà' generó entusiasmo y participación ('fent pinya'). La cena, comida popular catalana, y la actuación de un grupo de habaneras de Sitges completaron una velada muy agradable. Una última nota: para la organización, 4,2 sobre 5.

La siguiente: ESEC'97

Si todo va bien, será en Zurich (Suiza), del 22 al 25 de setiembre de 1997. Para información: 1) E-mail, esec97@ifi.unizh.ch
2) Internet: http://www.ifi.unizh.ch/congress/esec97.html
3) Escribiendo al organizador local: Martin Glinz, Dept. of Computer Science; University of Zurich Winterthurerstrasse 190 CH-8057 Zurich, Suiza.

Agradecimientos

En primer lugar a ATI, especialmente al amigo Pedro Gómez-Grau, por confiar en nosotros y darnos un marco. A CEPIS por el soporte y ayuda recibidos. A todos los comités, pero en especial a Wilhelm Schäfer y a Gregor Engels, con quienes he trabajado muy a gusto en todo momento. Y lo más importante, con quienes esta experiencia conjunta nos ha dejado una buena amistad. Y (last but not least) a Victor Obach y a toda su gente de DIFINSA por su saber hacer y estar: no es la primera vez que colaboramos y espero que habrá muchas más.

Saludo de Pedro Gómez Grau, Presidente de ATI, en la apertura de ESEC'95

No cabe duda que los profesionales del Software viven tiempos de confusión.

Si hace algunos años nos hubieran dicho que miles de *fans* pudieran hacer cola a las doce de la noche para comprar un Sistema Operativo junto con una *pizza* de regalo, seguro que hubieramos catalogado como loco a nuestro interlocutor. Y sin embargo es verdad, la industria del Software ha entrado en el negocio del espectáculo y la mayoría de los profesionales que viven del software se han enterado por la prensa.

Probable y desgraciadamente éste no es el único tema en que muchos profesionales se encuentran fuera de juego. En los últimos años se han producido en la tecnología del software cambios tan significativos que obligan a reciclarse profundamente a una cantidad importante de profesionales del software. Por ejemplo el paso de la artesanía a la ingeniería y la asunción del paradigma de la orientación a objetos no son cambios triviales y exigen un esfuerzo importante a aquellos que viven cada día de su producción de software.

Nuestra Asociación de Técnicos de Informática (ATI), la asociación de profesionales de la informática más importante del Estado Español, nació aquí, en Cataluña, hace 28 años, cuando el software tenía más de arte que de técnica. ATI es consciente de sus responsabilidades en estos momentos de cambio, contribuyendo en el marco de CEPIS al esfuerzo de organización de este Congreso Europeo como lo ha hecho siempre; y lo hará, dentro de sus posibilidades, en cualquier acontecimiento que contribuya a la mejora y a la divulgación de las tecnologías de la información.

Nuestra Asociación da la bienvenida a todos los asistentes a este congreso, en este hermoso marco y en la confianza de que sus trabajos contribuirán a pavimentar el camino donde nos movemos los profesionales hacia el futuro de la informática. Gracias a todos.

Ingeniería del Software

Métodos de Evaluación

Jose Luis Iparraguirre
Director de Desarrollo Corporativo
European Software Institute

Tanto los Gobiernos y Administraciones Públicas como la industria en general han redoblado sus esfuerzos para implantar la calidad del software. Cada vez es más común la experiencia del software que no cumple con las expectativas de los usuarios, con retrasos y sobrecostes que provocan gastos excesivos. A esto hay que añadir que los nuevos sistemas han incrementado drásticamente su tamaño y complejidad. Los pequeños sistemas de hoy son comparables a grandes sistemas de hace diez años y esta tendencia será más pronunciada a medida que la integración de aplicaciones se acelere y la potencia de los procesadores se multiplique.

La respuesta de los 90 está siendo el programa de Mejora de Procesos de Software. Durante muchos años la mejora de procesos se ha considerado en el mundo de la producción y el hardware como la mejor vía para incrementar la probabilidad de obtener un producto de calidad, ratificándose esta idea también en el mundo del software. Los esfuerzos, tanto en USA como en Europa, para consolidar las mejores prácticas de ingeniería para el perfeccionamiento de los procesos de software, han originado cierto número de modelos que pretenden: primero, determinar las fuerzas y debilidades en una organización; y después, aglutinar esfuerzos en base a conseguir acuerdos sobre lo que es un buen proceso.

Las cuatro principales iniciativas para la Mejora y Evaluación de Procesos de Software son: ISO 9001 y 9000-3, Capability Maturity Model (CMM), Bootstrap y SPICE. Son cuatro técnicas apropiadas para diferentes situaciones y entornos organizativos, pero no es posible compararlas punto por punto ya que, aunque todas ellas parten de técnicas de evaluación, Bootstrap se basa en un cuestionario con un modelo tácito, CMM y SPICE son modelos articulados, e ISO 9000 es un juego compacto de estándares. La comunidad del software es consciente de que ningún modelo satisfará todas las necesidades. Algunas compañías han empezado a combinar y abordar programas comparativos de estos métodos.

ISO 9000, por ejemplo, es muy útil en compañías que además de software fabrican equipos. Ofrece un estándar común (i.e., un lenguaje común) que permite la definición de procesos de calidad tanto en compañías de hardware como de software. En Europa se requiere con bastante frecuencia.

El ***'Capability Maturity Model'*** (CMM) del Instituto de Ingeniería del Software es, en la actualidad, el modelo más empleado y maduro, valga la redundancia. Diseñado para valorar sistemas de gran complejidad en cuanto a desarrollo de software, sus principales fuerzas son que facilita una visión completa del proceso de madurez organizacional e incluye mecanismos para una mejora continua de los procesos.

Bootstrap, enfocado a pequeñas y medianas empresas, puede valorar la madurez global de una organización, pero también se emplea para examinar procesos individuales de software y para valorar la conveniencia de determinadas tecnologías, así como el impacto de éstas en los procesos de desarrollo de software, en función del grado de madurez de la empresa.

SPICE, actualmente en fase de *beta-test,* combina elementos de ISO, CMM, y Bootstrap y está especialmente enfocado a estudiar el nivel de madurez de los procesos individuales, siendo su objetivo final la definición de un marco común de referencia en el que convivan el resto de los modelos enunciados anteriormente.

Breve introducción histórica

El Instituto de Ingeniería del Software (SEI) fue fundado por el Departamento de Defensa Norteamericano con el objeto de transferir los métodos de desarrollo y gestión de software más avanzados al entorno industrial americano. La primera contribución destacable del SEI, en el marco de dicha misión, fue el cuestionario publicado por Watts Humphrey y Bill Sweet en 1987, "A Method for Assessing the Software Engineering Capability of Contractors". El cuestionario Humphrey - Sweet tenía por objeto el facilitar a las empresas el proceso de autodiagnóstico de los puntos fuertes y débiles de su proceso producción de software, de manera que fuese utilizado como elemento de partida en la evaluación global de una empresa, apoyándose en un modelo de gestión implícito en la estructura del propio cuestionario.

El modo de trabajo se basaba en el envío del cuestionario, previamente a la llegada de un equipo de evaluadores, para su análisis por los responsables de distintos proyectos en la empresa. Posteriormente, el equipo de evaluadores utilizaría los resultados deducidos de las respuestas dadas a los elementos del cuestionario como punto de partida para las discusiones con el personal clave de la empresa. Las preguntas simplemente determinaban si existían ciertas prácticas de Ingeniería del Software, ofreciéndose la posibilidad de responder afirmativa o negativamente a dicha práctica. La idea subyacente no era la de valorar a la empresa en base al cuestionario, sino utilizarlo para determinar qué áreas deberían ser exploradas en el proceso de evaluación propiamente dicha, que sería conducida de acuerdo con el modelo de referencia implícito de procesos de software del SEI.

En el mismo año (1987) en que Humphrey y Sweet publicaron su cuestionario de Evaluación, la Organización Internacional de Estándares desarrolló ISO 9000 con el fin de crear una serie de normas para los productos objeto de comercio internacional y más específicamente dentro de la Comunidad Económica Europea. Los estándares ISO provenían indirectamente, a través de la OTAN y la Institución Británica de Estándares, de los programas estándares de gestión de calidad del Departamento de Defensa de los Estados Unidos, por lo que en cierto modo subyace una cultura común en ambos métodos. La serie de estándares ISO 9000 se compone de una colección

de documentos relacionados que, combinados, constituyen un sistema de calidad. El fin asociado a esta colección de documentos será que una gran mayoría de empresas y agencias Europeas obtengan la certificación ISO 9000, constituyéndose de dicha forma en un substrato de base a una cultura de calidad centrada en el marco del contrato cliente proveedor.

El éxito alcanzado en el sector industrial por la serie de estándares ISO 9000 hacen de éstos el sistema de gestión/monitorización más extendido a nivel mundial, habiendo sido adoptado por miles de empresas en 60 países de todo el mundo como los miembros de la Unión Europea, Canadá, Japón, y EUA.

En 1990 el Programa Estratégico Europeo para las Tecnologías de la Información ESPRIT tomó como elemento de partida los distintos modelos preexistentes y financió el desarrollo de la metodología Bootstrap. Bootstrap fue desarrollado por un consorcio formado por empresas originarias de Alemania, Italia, Finlandia, Austria y Bélgica. El objetivo principal del proyecto era la integración de los conceptos preexistentes en Estados Unidos y a nivel internacional en la realidad industrial europea, marcada esencialmente por la abundancia de PYMEs especializadas en pequeños nichos de mercado con impacto geográfico local. Mientras que el Modelo de Madurez del SEI estaba principalmente orientado al sector Aeroespacial y de Defensa con aplicaciones de gran escala y extremadamente complejas, Bootstrap se orientaba a las aplicaciones comerciales europeas, centrándose en sistemas reducidos y clasificando los componentes individuales de los procesos de producción de software (contemplando la tecnología como un *cluster* más de atributos a considerar y como aspecto integrado en el desarrollo de software, no contemplado anteriormente por el modelo de Madurez del SEI).

La metodología Bootstrap se basa, de manera práctica, en un cuestionario (confidencial), que cubre el conjunto de componentes y practicas de Ingeniería del Software que una empresa debería poner en práctica a lo largo de las distintas fases del ciclo de vida de un desarrollo y que servirá como elemento de partida para la creación de un plan de mejora de las prácticas empleadas o en su caso ausentes. El modelo subyacente se basa en el cuestionario del SEI de 1987,en el modelo del ciclo de vida de la Agencia Europea del Espacio ESA-PSS-05-0 y en ISO 9000-3, de manera que Bootstrap podría considerarse como un paso natural hacia la convergencia e integración de los conceptos preexistentes. Como consecuencia del desarrollo del método, en 1994 se fundó el Instituto Bootstrap, con sede en Bruselas, para la explotación del método y gestión de su evolución, de manera que integre sucesivamente el conjunto de avances que se deriven de la actividad en el sector.

SPICE es el último en esta serie de métodos para la evaluación de los procesos de producción de software y su consiguiente mejora. El proyecto SPICE se inició oficialmente en junio de 1991 en la asamblea plenaria conjunta del Comité Técnico del ISO e IEC (Comité Electrotécnico Internacional). En enero de 1993 ISO creó formalmente SPICE, proyecto para 'determinación de la madurez y mejora de los procesos de software' *(Software Process Improvement and Capability Determination)*. SPICE se encamina al desarrollo de estándares en los procesos de evaluación de la producción de software, basados por un lado en los desarrollos preexistentes y por otro en el consenso de la comunidad internacional envuelta en actividades de Ingeniería del Software, de manera que SPICE sea un marco común en el que el resto de métodos preexistentes tuviese su cabida, sirviendo como elemento de referencia común.

Los participantes en SPICE proceden de todo el mundo (más de 200 empresas de cuatro continentes, en un esfuerzo común por definir pr‡cticas contrastadas para la Gestión y el Desarrollo de software, que por un lado sirvan para la evaluación de la madurez de la empresa (aspecto que interesaría a quién está en la situación de seleccionar un proveedor) y por otro para la preparación de planes de mejora que permitan aumentar la competitividad de la empresa, en función de sus objetivos estratégicos, reduciendo los plazos de desarrollo y mejorando la productividad y la calidad del producto final.

Cuatro centros técnicos velan por la coherencia de las labores llevadas a cabo y coordinan los esfuerzos en sus distintas zonas de influencia: el Instituto Europeo del Software (Zamudio, España) para Europa, el Instituto de Ingeniería de Software (Pittsburgh) en Estados Unidos de América , Bell Canada (Quebec) en Canada y la Universidad de Griffith en Australia.

Actualmente SPICE se encuentra en una fase piloto de pruebas *(beta test)* para probar tanto la cobertura del estándar como su aplicabilidad en un amplio rango de tipos de empresas, entornos culturales e idiomáticos, de manera que el resultado del proyecto sea una norma de amplia aceptación desde sus orígenes.

El conjunto de documentos que componen el proyecto se someterán al proceso de estandarización ISO en 1996 y cubren tanto cuestiones relacionadas con los procesos de producción de software como con las personas, la tecnología, las prácticas de gestión, las relaciones cliente-proveedor (satisfacción del cliente), la gestión de la calidad, el desarrollo de software y los procesos de mantenimiento.

Conclusión

Los puntos enumerados en los párrafos anteriores muestran el interés despertado, en los últimos diez años por los aspectos relacionados con la Ingeniería del Software y la modelización de los procesos de desarrollo del mismo, con esfuerzos dirigidos a la materialización de métodos que ofrezcan un panorama claro para la mejora de la competitividad de las empresas embarcadas en la puesta en operación de planes de mejora de su procesos de producción de software. De la propia proliferación de métodos puede deducirse la no universalización de los mismos; de ahí que no sea posible la identificación de una única 'bala de plata' para la globalidad del problema, en términos de tamaño de la empresa, área de actividad, marco cultural de la misma, madurez actual de sus procesos, etc. Es necesario e incluso deseable identificar el método de evaluación/mejora que más se adapte a cada caso.

Esta diversidad de métodos no debe conformar una maraña de ramas que con su espesura no nos permitan ver el bosque. Los métodos de evaluación/mejora de procesos de producción de software no serán otra cosa que el medio por el que accedemos al fin perseguido con estas disciplinas de la Ingenier'a del Software: mejorar la madurez de nuestra empresa, de manera que hagamos factible el trinomio subyacente de producir más barato, más rápido y con más calidad, con independencia del método empleado para llegar a este fin.

Ingeniería del Software

AgOra: una arquitectura orientada a agentes para la modelización de sistemas de información

José Cuevas *, Jaume Devesa *, Òscar Coltell**, Isidro Ramos*
* Dpto. de Sistemas Informáticos y Computación, UPV
**Dpto. de Informática, Universidad Jaume I de Castellón
e-mail: jcuevas@dsic.upv.es

Resumen: *los avances en la modelización de sistemas de información han aportado nuevas metodologías, a fin de capturar el sistema a estudiar de manera más cercana a la realidad; lo que viene a llamarse 'estrechar el gap semántico'. Nuestra última propuesta consiste en una arquitectura orientada a agentes, con un nuevo lenguaje llamado AgOrA (Agent-Oriented Architecture) como soporte lingüístico, que ofrece una visión del sistema de información como una colección de agentes. El concepto de agente es una especialización del concepto de objeto, que tiene una actividad propia, la cual afectará la de otros agentes.*

1. Introducción

Nuestro esfuerzo durante los últimos años en la modelización de sistemas de información ha dado como resultado diferentes propuestas de lenguajes de especificación: cada uno ha tenido como características principales las distintas inquietudes que en este tema han ido surgiendo en cada momento. Desde el primitivo PROTOLOG hasta el reciente AgOrA, la intención ha sido proponer una notación formal que modelice el sistema de información de manera más próxima al Espacio del Problema.

A fin de entender el porqué del nacimiento de AgOrA y los conceptos que lo soportan, daremos en el **apartado 2** un repaso a las propuestas anteriores, que todas ellas ilustran la evolución acontecida en este campo en los últimos tiempos. En el **apartado 3** estudiaremos la metodología de análisis orientado a agentes que surge de aplicar estas ideas y la compararemos con la metodología de análisis orientado a objetos para fijar las diferencias entre ambos. En el **apartado 4** describiremos la sintaxis del lenguaje AgOrA, que es una de las partes del soporte lingüístico de esta metodología -siendo la otra la semántica. Por último, en el **apartado 5**, expresaremos unas conclusiones que intentarán resolver las últimas dudas.

2. Antecedentes de lenguajes de especificación

2.1. Dando cuenta del cambio de estado

En 1989 se idea el sistema PROTOLOG, presentado en [GACPRR89] como un generador de prototipos para validación de modelos conceptuales de sistemas de información. A partir de una especificación formal, que define un modelo dinámico de estados múltiples, dicho sistema genera una base de datos en forma de programa Prolog. El lenguaje de especificación permite definir una serie de tuplas, llamadas **relaciones básicas**, que son las entradas al sistema (eventos). Otro conjunto de tuplas, las **relaciones derivadas**, representan la salida de información. Aquéllas provocan cambios en éstas de acuerdo a un conjunto de reglas de derivación. Además, pueden existir restricciones que en todo momento deben de validarse.

En este primer paso adquiere una importancia clave el concepto de **estado** del sistema, cuyas componentes son el conjunto de relaciones derivadas. Estas relaciones cambian con el tiempo y el **cambio de estado** está provocado por la ocurrencia de tuplas de eventos de acuerdo a las reglas de derivación, obteniéndose tuplas de las relaciones derivadas. Además, el conjunto de estados posibles se restringe mediante restricciones.

2.2. El modelo orientado a objetos

Unos años más tarde, la influencia del lenguaje OBLOG en los artículos de Sernadas [CSS89] [SSG+91] encamina las investigaciones sobre este tema al mundo de los objetos. Fruto de ello es el desarrollo del sistema MicroOblog ([Ramos90]), en sus dos versiones R-MOL y F-MOL, según se base su expresividad en la lógica clausal (relacional) o funcional. Siguiendo el modelo orientado a objetos, el sistema a analizar se fragmenta en diferentes unidades de estudio, cada una de las cuales es un **objeto**. Este concepto encapsula un estado y un comportamiento. El estado de un objeto viene definido por los valores de sus componentes, llamadas **atributos**. Un conjunto de operaciones, llamadas **eventos**, representa su comportamiento. Un evento puede modificar el valor de un atributo, y por consiguiente el estado del objeto, de acuerdo a unas reglas que son fórmulas bien formadas de una lógica de primer orden, según la expresividad elegida. El conjunto de estados posibles se determina mediante **restricciones** sobre valores de los atributos, o mediante **precondiciones** sobre los eventos.

Este primer intento de lenguaje de especificación en el mundo de la orientación a objetos permite describir las características (atributos, eventos, restricciones y precondiciones) comunes a una clase de objetos, en forma de plantilla o *template*. De esta forma, se puede modelizar el sistema de información como un conjunto de objetos que tienen un estado local oculto y que ofrecen ***servicios*** al entorno. Éstos son de dos tipos: consulta del valor de un atributo, que permite observar una componente del estado del objeto; o bien ejecución de un evento, que permite provocar un cambio en el estado del objeto. De momento, un objeto es un mero ente pasivo, que ofrece servicios al entorno. Un cambio de estado es una respuesta del objeto a una demanda del entorno.

2.3. Añadiendo actividad al sistema

En MicroOblog la actividad era producida única y exclusivamente por el entorno (el usuario del sistema de prototipación), activando los eventos que producen cambios de estado en los objetos. Así pues, el siguiente paso es obligado: hacer que un objeto sea activo. El afán por modelizar sistemas activos nos lleva a diseñar el lenguaje OASIS ([Pastor92]) con tres versiones: R-OASIS, con una expresividad clausal; F-OASIS, con una expresividad funcional; y L-OASIS, con una expresividad clausal con igualdad. La principal incorporación es el concepto de disparo o ***trigger*** como una fórmula que permite a un objeto activar eventos de sí mismo, de otro objeto o de toda una clase. En los *triggers* se utiliza una sintaxis 'a la POLKA' [Davison87], con fórmulas de la forma siguiente:

destino :: evento :- condición

destino (una de las tres palabras clave *self, object, class)* indica que el evento provocará un cambio de estado en sí mismo, en un determinado objeto o en toda la población de una clase, si se cumple la condición especificada. De esta forma, un objeto es, además de servidor, un primitivo solicitante de servicios.

2.4. Los objetos como clientes y servidores

Profundizando en la idea anterior, nos encontramos con la necesidad de especificar la vida de los objetos como ciclos de demandas de servicios a otros objetos y de ofertas de servicios demandados. Así, se incluye en la especificación de la clase un término de un álgebra de procesos (o *process query*) construido a partir de los eventos conectados con operadores de secuencia, alternativa, etc. Se introduce un nuevo concepto: el de **acción**, como la demanda de un objeto para que en otro acontezca un evento. Se fija además la semántica que subyace a OASIS. Se trata de la **lógica dinámica** de [Harel84], que explica a la perfección el cambio de estado de los objetos como respuesta a una demanda de un servicio por parte de otro objeto. La lógica dinámica define el cambio de estado en términos de una estructura de Kripke, dando una semántica de mundos posibles. El modelo es una tupla (W,t,r), donde el primer elemento es un conjunto de estados, y los otros dos son funciones. Informalmente, asigna a cada fórmula un subconjunto de W, que es el conjunto de estados en los que la fórmula es cierta; a su vez, la función asigna a cada proceso una relación binaria entre estados de W que define la transición entre estados que provoca la ejecución de cada proceso. Todas estas nuevas ideas tomadas de [Wieringa90] y [WM92] se han incorporado a OASIS v2 ([RPCD93]), una nueva versión del lenguaje de especificación. En él, se ha adaptado también la sintaxis de las fórmulas bien formadas de acuerdo a la nueva expresividad basada en la lógica dinámica.

2.5. El concepto de agente

El último paso ha sido la revisión del concepto de objeto en su aspecto activo, lo que nos ha conducido al concepto de agente que Dubois y su equipo presenta en sus trabajos [DDBP93] [DuBois795] [DuBois895]. Un **agente** es una especialización del concepto de objeto. Los agentes son entes con actividad, es decir, realizan un determinado conjunto de tareas y emplean un tiempo en desarrollarlas. Llamamos a estas tareas **acciones** y son ejecutadas por los agentes sin fijarse en que, a causa de ellas, acontecen cambios de estado en otros agentes. Es decir, mientras que en OASIS la actividad de los objetos está encaminada a provocar cambios de estado, ahora dicha actividad tiene otros fines, aunque como efecto lateral pueda producirlos al ser percibida por otros agentes. O dicho de otro modo, un cambio de estado en un agente es una reacción al **percibir** la actividad de otro, y no una respuesta del primero a una solicitud expresa por parte del segundo. Se trata de una visión dual del sistema de información que rompe con el concepto de objeto como suministrador de servicios y que permite describir la actividad del sistema en su conjunto, independientemente del efecto que produce dicha actividad. Así pues, se puede modelizar el tiempo que emplea un robot en mover su brazo mecánico, o la espera que se produce en un cajero automático desde que se solicita una cantidad de dinero hasta que ésta es suministrada. Fruto de estas ideas es el lenguaje **AgOrA**, que presentamos en este artículo. AgOrA permite especificar clases de agentes, dando una plantilla de sus características.

3. AgOrA versus ObOrA

Las ideas que presentamos sugieren una nueva metodología de análisis, basada en el concepto de agente, diferente de las metodologías de ObOrA (análisis orientado a objetos). En este apartado compararemos ambas con un ejemplo: el funcionamiento de un cajero automático, donde el cliente de un banco puede extraer o ingresar dinero. Este ejemplo está definido completamente en el **anexo** de este artículo.

En AgOrA, el punto inicial es identificar los agentes que componen el sistema de información. Los agentes pueden coincidir con los objetos que se obtendrían aplicando una metodología de ObOrA, pero esto no es así necesariamente. De hecho, el proceso que se sigue para identificarlos es diferente. Un agente es toda entidad del sistema con actividad. En nuestro ejemplo, son agentes el ***cliente,*** que es la persona que utiliza el cajero para sacar dinero; el ***cajero,*** que es la máquina expendedora de dinero; y el ***banco,*** que es la entidad que debe supervisar los movimientos del cajero. Todos ellos podrían coincidir con los objetos de un ObOrA. Seguidamente, para cada agente se identifican sus atributos, o componentes de estado, y sus acciones. Es importante reseñar qué otros agentes perciben esos atributos y esas acciones.

Para el cliente, las acciones son *introducir la tarjeta, teclear el número personal, elegir la operación a realizar, teclear el importe requerido, introducir el dinero en caso de un ingreso, retirar la tarjeta, retirar el comprobante y retirar el dinero;* la ejecución de todas ellas es percibida por el cajero, si éste no está fuera de servicio. Comparativamente, cabría reseñar que ObOrA llamaría eventos a las acciones, y las desplazaría al ámbito de los objetos que las perciben; es decir, se trataría de eventos del cajero activados por el cliente.

Siguiendo la metodología AgOrA, no se han de especificar atributos en el cliente, porque su estado no es percibido por ningún otro agente. Un ObOrA nos diría que el saldo de la cuenta corriente es un atributo del cliente, que se modifica cuando ocurren todas las acciones anteriores. Pero hacerlo así es un error, pues realmente es el banco quien actualiza dicho saldo; por lo tanto se trata de un atributo del banco.

Para el cajero, los atributos ofrecen información sobre si el cajero está fuera de servicio y sobre el dinero disponible. Otros almacenarán información sobre la operación actual: la tarjeta, el número personal y el importe introducidos, la operación elegida, o si la transacción es aceptada o no. Sólo el primer atributo es percibido por el cliente. Sus acciones serían *solicitar la introducción de una tarjeta, solicitar la introducción del número personal, solicitar la elección de la operación a realizar, solicitar la introducción del importe, solicitar la introducción del dinero a ingresar, validar la transacción, escupir la tarjeta, escupir el comprobante, escupir el dinero solicitado.* Todas estas acciones son percibidas por el cliente; y la última es percibida además por el banco, a fin de modificar el saldo de manera oportuna.

En ObOrA no se identificarían estas acciones, porque unas tienen como respuesta otras acciones del cliente, las cuales ya se habrían declarado a su vez como eventos del cajero; y en el caso de validar la transacción, se trata de modelizar la espera que se produce en el cajero, y no un servicio ofertado. Por todo ello, no habría eventos que correspondieran con las acciones del cajero.

Como se ha indicado antes, el banco guarda la información sobre las cuentas corrientes: el saldo, los permisos para realizar los diferentes tipos de operaciones, los números personales, etc.; todo ello en forma de atributos accesibles por el cajero. Otro atributo del banco indica si éste está accesible o no. Una de las acciones que realiza el banco es *modificar el saldo* de una componente de dicha tabla, cuando perciba que el cajero ha realizado la acción *escupir el dinero solicitado* o el cliente ha *introducido el dinero a ingresar*. Caso de no estar el banco accesible, no podrá percibir estas acciones.

En resumen, en AgOrA hay un desplazamiento de la declaración de las acciones al agente que las ejecuta. De esta forma, se construye un modelo más real de la actividad de los agentes [AR92]. Además, el concepto de percepción modeliza la actividad del todo el sistema. En nuestro ejemplo, la interfaz entre cliente y cajero, y la comunicación entre cajero y banco, se han especificado por medio de la percepción, por parte de uno de ellos, de las acciones y el estado del otro. Dicha percepción es dinámica, lo cual quiere decir que un agente puede dejar de percibir el estado o la actividad de otro, según determinadas condiciones que veremos en el punto siguiente.

4. La sintaxis

4.1. Definición de una clase de agentes

La definición de una clase de agentes está constituida por tres elementos: el nombre de la clase, la identificación *(aid, agent-identifier)* y la plantilla o *template*.

- En el ***nombre de una clase*** se utiliza una elemento de un *sort* (TAD) -posiblemente una cadena de caracteres- que represente significativamente la clase de agentes a modelar.

- Los mecanismos de ***identificación*** propuestos son de tres tipos: *aid interna* que pertenece a un sort (TAD) determinado; *aid externa,* dependiente del espacio del problema; y *adopción de una clave,* similar al modelo relacional, que relacione unos agentes con otros. La identificación de agentes se puede supeditar a la identificación de clases de agentes -abstracciones de orden superior al concepto de agente- que pueden constituir lo que podría denominarse *meta-agentes*.

- El apartado siguiente desarrolla la declaración de una ***plantilla.*** En la plantilla de una definición de clase de agentes en AgOrA (**figura 1**), se incluye un conjunto de palabras reservadas (**agent_class, identifier is, agent_template**) y otro conjunto de palabras que representan los elementos variables de la definición y que tienen el significado siguiente:

ag_name :	nombre de un agente.
ag_tp_name :	nombre de la plantilla de la clase de agentes.
<TAD>	sort al que pertenece el identificador de la clase de agentes.

4.2. La Declaración de una plantilla o template

El lenguaje AgOrA permite declarar las propiedades de una clase de agentes y la percepción que tienen éstos del exterior. La declaración de una plantilla o template tiene tres partes:

```
agent_class ag_name
identifier is <TAD>
agent_template ag_tp_name
```

Figura. 1. Definición de una clase de agentes en AgOrA

- **signatura:** consta de la declaración de las componentes de estado (atributos) y de las acciones que realiza el agente;
- **restricciones**: constan de fórmulas temporales que restringen el conjunto de estados válidos, evaluaciones de los atributos, obligaciones y prohibiciones sobre las acciones, etc.;
- **percepciones:** indican cuándo el agente no percibe la actividad y el estado de otros agentes, y cuándo otros agentes no percibe la actividad y el estado del que se está especificando.

En la plantilla de una especificación AgOrA (**figura 2**), que se aplica en la declaración de un *template*, se incluye un conjunto de palabras reservadas (**agent_template, attributes, default, derived from, exported to, actions,** etc.) y otro conjunto de palabras que representan los elementos variables de la declaración, y que tienen el significado siguiente:

ag_tp_name :	nombre de la plantilla de la clase de agentes
at_name :	nombre de un atributo
ac_name :	nombre de una acción
arg :	argumento de una acción, sea variable, constante o una expresión
type :	tipo donde toman valores los atributos y argumentos.
constant :	constante de un determinado tipo de datos
expression :	expresión cuyo valor pertenece a un determinado tipo de datos
formula :	fórmula bien formada con términos que expresan relaciones temporales, comparaciones de valores y cuantificación
process :	término del álgebra de procesos.
time :	valor medido en unidades de tiempo.

La expresión de la situación que permite aplicar las restricciones está representada en la palabra reservada **situation** (**figura 3**).

```
agent_template ag_tp_name
attributes
   at_name : type      default constant
                       derived from expression
                       exported to ag_name[, ag_name[, ...]]
actions
   ac_name (arg[, arg[, ...]]) duration time priority constant
                               composed of process
                               exported to ag_name[, ag_name[, ...]]
check
   formula
evaluate
   at_name = expression      situation
cause
   ac_name (arg[, arg[, ...]]) situation
force
   ac_name (arg[, arg[, ...]]) situation
prevent
   ac_name (arg[, arg[, ...]]) situation
ignore
   name
   name(arg[,arg[,...]])     situation
hide
   name
   name(arg[,arg[,...]])     ag_name[, ag_name[, ...]]     situation
```

Figura 2: Plantilla de una especificación AgOra

El término 'formula' es una restricción de un Lenguaje de Primer Orden con la posibilidad de incluir operadores temporales, tales como **always, sometimes, in the past, in the future, until** y **since**. Aunque no se acompañe de la palabra reservada **ends**, se controla por defecto cuando termina una acción, salvo que se indique lo contrario, con la palabra reservada **begins**.

4.2. La Signatura

Como hemos comentado, la signatura consta de dos partes: declaración de atributos y declaración de acciones. Previamente, pueden haber una **declaración de las variables** que se utilicen a lo largo de toda la especificación del agente, y que son locales a ella. Se dará el nombre y el tipo de la variable. Los tipos utilizados pueden ser los habituales (entero, booleano, string, etc.) que estarán predefinidos, u otros que se podrán especificar mediante un lenguaje algebraico. Por ejemplo,

```
o:operación;
```

da cuenta de una variable de tipo operación, que puede ser definida como un conjunto con tres posibles valores: *reintegro, ingreso, consulta de saldo*. Los nombres de los agentes declarados también son tipos válidos, cuyas variables representan instancias de la clase de agente correspondiente.

En la **declaración de atributos** se especifican las componentes del estado del agente. Para cada una de ellas básicamente se da su tipo y el/los agente/s que pueden percibirla. En nuestro ejemplo, el importe solicitado es un atributo del cajero, pertenece a los números naturales y puede ser percibido por el banco:

```
imp: nat exported to banco;
```

Existe la posibilidad de dotar de un valor inicial a un atributo. Tendrá así dicho valor desde el principio de la vida del agente hasta que ocurra un cambio de estado que afecte al atributo. Por ejemplo, si el dinero disponible del cajero será inicialmente de un millón de pesetas:

```
disponible: nat default 1000000;
```

Una característica más es que el valor de un atributo puede depender del valor de otros. He aquí uno de tipo booleano que depende del valor de otros cuatro del mismo tipo, y cuyo valor se calcula mediante una expresión lógica:

```
ok_trans: bool derived from ok_tarj and ok_pin and ok_op and ok_saldo;
```

Los atributos *disponible* y *ok_trans* no se exportan, lo cual quiere decir que ningún otro agente podrá percibirlos: son locales al cajero. Un ejemplo más completo es el atributo que indica si el cajero está en servicio o fuera de él: es percibido por los agentes cliente y banco, y su valor depende en todo momento (incluido el inicial) del dinero disponible del cajero:

```
funciona: bool derived from disponible < 10000
          exported to cliente, banco;
```

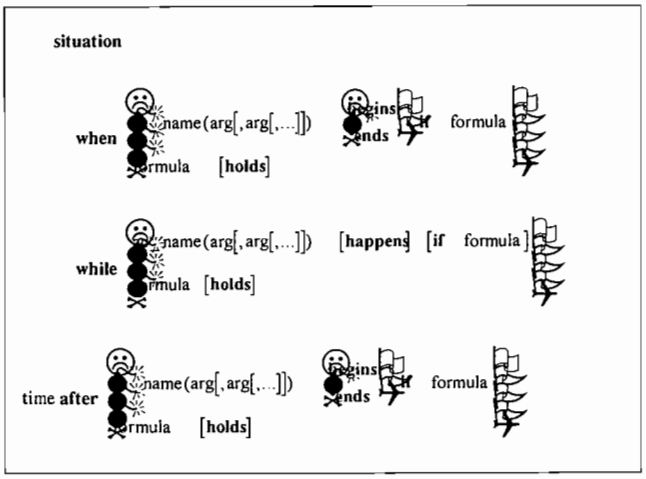


Figura 3: Combinaciones posibles de la especificación de 'situación'

La **declaración de acciones** expresa las diferentes formas de actividad del agente. Cada acción tiene una duración y puede ser percibida por otros agentes. Así pues, la acción por parte del cajero de escupir el dinero solicitado se realiza en 2 segundos y es percibida por el cliente (pues deberá retirarlo) y el banco (quien deberá actualizar el saldo):

```
escupe_din() duration 2 seconds exported to cliente, banco;
```

Algunas acciones pueden estar compuestas por otras más sencillas. Esto se expresa mediante un término del álgebra de procesos formado por acciones y por las operaciones ';' (secuencia), '+' (alternativa), etc. según se detalla en [RPCD93]. En el ejemplo, el banco puede anular una cuenta correspondiente a una tarjeta, que consiste en revocar el permiso para hacer reintegros e ingresos, seguido de la cancelación de la tarjeta; todo ello lleva un tiempo de 5 minutos. Esta acción es local al banco y no puede ser percibida desde el exterior:

```
anular_cuenta(t) composed of revocar_perm(t,R)
                             revocar_perm(t,I) cancelar_tarj(t);
                 duration 5 minutes;
```

Como vemos, las acciones pueden o no tener argumentos. En la declaración anterior, supondremos que existe una variable *t* de tipo *tarjeta*. Este tipo consistirá en una tupla formada por el código de tarjeta, el código de cuenta corriente, y la fecha de caducidad. También la acción de *revocar un permiso* por el banco tiene argumentos: la tarjeta y la operación que ya no se le permitirá realizar.

Finalmente, las acciones pueden tener una prioridad asignada en el momento de la declaración, que puede ser un valor, o por defecto (para todas las acciones que no tengan especial relevancia a priori), o que determine acciones de alta prioridad.

4.3. Las Restricciones

Las restricciones son fórmulas de diferentes tipos, con diferentes significados. Cada una corresponde a una sección de la especificación encabezada por una palabra reservada: un verbo en inglés que expresa lo que hay que hacer. Todas ellas se utilizan para restringir las posibles trazas o vidas de los agentes. Puede hacerse de diferentes formas:

- restricciones de integridad dinámicas, que restringen el conjunto de secuencias de estados;
- evaluaciones, que permiten conocer el estado al que se llega después de la ejecución de una acción;
- relaciones de causa-efecto entre acciones (causalidades), que establecen la ocurrencia de una acción tras la ejecución de otra;
- obligaciones, que establecen la ocurrencia de una acción si una determinada condición tiene lugar;
- prohibiciones, que restringen la ocurrencia de acciones en determinadas condiciones.

Las **restricciones de integridad dinámicas** usan operadores temporales para expresar las restricciones que deben chequearse (de ahí la palabra reservada **check**). Así debe comprobarse que alguna vez el dinero disponible del cajero supere 50.000 pesetas después de que haya estado fuera de servicio:

check sometimes in the future disponible > 50000
since funciona = FALSE;

Las **evaluaciones** dicen cómo cambia el valor de un atributo tras la ejecución de una acción. Si esta acción es ejecutada por un agente distinto, se calificará con una variable de tipo agente. Así, en el caso de un ingreso, el disponible del cajero se incrementa en el importe seleccionado cuando el cliente introduce el dinero (supondremos declarada la variable *cl* de tipo *cliente*):

evaluate disponible = disponible + imp **when** cl.intro_dinero() **ends**;

A la fórmula de evaluación puede añadirse una condición que, según se cumpla o no, discriminará diferentes valores del atributo. Así, tras validar el saldo, el atributo *ok_saldo* será cierto si la operación no es un reintegro o bien, en caso que lo sea, haya suficiente saldo en la tarjeta (atributo del banco de tipo tabla, indexada por la tarjeta). Será falso en caso contrario:

evaluate
ok_saldo=TRUE **when** valida_saldo() **ends**
if op <> R or op = R and b.saldo[tarj] >= imp;
ok_saldo=FALSE **when** valida_saldo() **ends else**;

Las **causalidades** establecen relaciones entre la ejecución de acciones y en nuestro ejemplo explican muy bien el comportamiento del cajero. Cuando el cliente introduce la tarjeta, el cajero solicita el número personal (pin), y esto hace que el cliente lo teclee, etc. En el cajero tendremos (*c* y *t* son variables de tipo *cajero* y *tarjeta* respectivamente):

cause solicita_pin() **when** cl.intro_tarj(c,t) **ends**;

mientras que en el cliente tendremos (*c* y *p* son variables de tipo *cajero* y *string* respectivamente):

cause intro_pin(c,p) **when** c.solicita_pin() **ends**;

Como en las evaluaciones, se pueden expresar condiciones y también el tiempo que debe pasar entre una acción y otra. Tras retirar el cliente la tarjeta y el comprobante, el cajero escupirá el dinero sólo en caso de que la operación sea un reintegro y el resultado de validar la transacción sea correcto. Si esa condición no se cumple, el cajero pasará directamente a solicitar una nueva tarjeta, tarea que realizará después de 3 segundos:

cause
escupe_din() **when** cl.retira_comp() **ends**
if op = R and ok_trans = TRUE;
solicita_tarj() 3 seconds **after** cl.retira_comp() **ends else**;

Las **obligaciones** fuerzan (por eso la palabra reservada **force**) a que se ejecute una acción si se da una determinada condición. Opcionalmente se puede dar un tiempo de reacción, como en el caso anterior. En el ejemplo, el cajero se detiene si el dinero disponible es menor de 10.000 pesetas:

force detiene_cajero() **when** disponible < 10000 **holds**;

Las **prohibiciones** previenen (en inglés **prevent**) la ejecución de acciones, según determinadas condiciones. Este es el caso del cajero, que no podrá solicitar una nueva tarjeta si el cajero está fuera de servicio:

prevent solicita_tarj() **while** funciona = FALSE **holds**;

4.4. Las Percepciones

Ya se indicó que la percepción es dinámica, o sea: un agente puede percibir en un instante dado que otro está realizando una actividad o bien que un atributo tiene un determinado valor, pero es incapaz de percibirlo en otro momento. Esto se consigue de dos formas: ignorando u ocultando. Veamos cada una de ellas.

Cuando un agente exporta un atributo o una acción a otro, por defecto el segundo puede percibirlos en todo momento. Sin embargo, es posible hacer que los **ignore** ante determinadas condiciones. Por ejemplo, el cajero percibe el instante en que el cliente introduce la tarjeta, excepto si está fuera de servicio. Es este caso, cualquier intento de comenzar una operación con el cajero es ignorada por éste:

ignore cl.intro_tarj(c,t) **while** funciona = FALSE **holds**;

Además, un agente puede **ocultar** (en inglés **hide**) un atributo o una acción a otro, a pesar de que se los haya exportado, si se cumplen ciertas condiciones. Es el caso del banco, que oculta el saldo de la tarjeta al cajero cuando se encuentra inaccesible por la red, simulando la imposibilidad del cajero de conectar con él:

hide saldo **to** cajero **while** accesible = FALSE **holds**;

5. Conclusiones

El lenguaje presentado constituye la base de una metodología de análisis diferente de las metodologías orientadas a objeto al uso (ya bastante extendidas), porque recoge unos aspectos de la realidad que no contemplan estas metodologías: el comportamiento activo de los componentes de un sistema y su interacción con el resto de componentes del mismo sistema, que constituye el modelo del sistema. Con ello pretendemos modelar sistemas cuyo comportamiento tiene una fuerte componente temporal y cuyos elementos no funcionan necesariamente en el modo secuencial característico propio de sistemas que funcionan en ambientes industriales.

Podemos afirmar que, en el caso de herramientas CASE, se tienen muy claras todas las fases a seguir a la hora de realizar un desarrollo de un Sistema de Información, desde la especificación formal al mantenimiento. En estos entornos se cuidan de forma precisa todos los aspectos relativos al software, pero ninguno relativo al entorno sobre el que se va a utilizar. En este artículo hemos presentado una primera versión de un lenguaje orientado a agentes dentro de lo que sería un sistema de composición de requisitos del entorno, en lugar de requisitos del software. Nuestra intención es ampliar los trabajos realizados durante muchos años sobre el entorno CASE y abrir una rama nueva dentro del entorno CIM (Computer Integrated Manufacturing) mediante un lenguaje que se caracterizará por:

- ***su expresividad:*** los requisitos en las acciones, percepciones y rendimiento general del sistema se estructuran en términos de agentes.
- ***su grado de formalidad:*** las distintas restricciones se expresan en términos de variantes de la Lógica de Primer Orden, como la Lógica Temporal, la Lógica Deóntica etc. En la actualidad se están estudiando otros sistemas formales.

Las operaciones entre las clases constituyen una de las características más interesantes y que mayor poder expresivo aporta a los entornos antes mencionados. Entendiendo que las operaciones entre clases son esenciales en cualquier entorno de especificación, es nuestra intención, en primer lugar, el estudio de operaciones entre agentes, teniendo como primer objetivo el estudio de las operaciones de agregación y herencia imprescindibles en cualquier lenguaje de especificación.

La experiencia demuestra que la ingeniería del software no es efectiva si no existen herramientas que la soporten. A medio plazo, queremos desarrollar un entorno integrado de la ingeniería de los requisitos. En principio, este entorno debería tener:
- Facilidades de edición por medio de un ***editor*** que nos permita trabajar de forma textual y gráfica.
- Facilidades de validación por medio de un ***animador*** que nos sirva para ver si los requisitos iniciales del cliente han sido correctamente capturados por el analista.

A corto plazo tenemos dos objetivos determinados: realizar la descripción formal de la semántica del lenguaje, de forma que mejoremos la calidad en cuanto a los aspectos de correción y completitud; y la realización de un prototipo del entorno orientado a agentes.

6. Referencias

[AR92] Astesiano E., Reggio G.; *Algebraic Specification of Concurrency.* Technical Report, DISI, Univ.of Genova, 1992.

[CAN90] Canós Cerdá J.H.; *Implementación de un Monitor de Restricciones de Integridad Dinámicas expresadas en Lógica Temporal;* trabajo de doctorado, DSIC-UPV, Valencia, 90

[CF82] Casanova M.A., Furtado A.L.; *A Family of Temporal Languages for the description of Transition Cosntraints,* in *Procs. of the Workshop on Logical Bases for Data Bases,* Toulouse, 1982.

[CSS89] Costa J-F., Sernadas A., Sernadas C.; *OB-89 User's Manual.* Internal Report, INESC, Lisbon, 1989.

[DDBP93] Dubois E., Du Bois P., Petit M.; *OO Requirements Analysis: an Agent Perspective.* Proc of the 7th Conference on OO Programming - ECOOP 93, LNCS 707, Springer-Verlag, 1993:458-481.

[DuBois795] Du Bois P.; *Intiutive Definition of the Albert II Language.* Internal Report, RP-95-007, Institut d'Informatique, FUNDP, Namur, 1995.

[DuBois895] Du Bois P.; *Semantic Definition of the Albert II Language.* Internal Report, RP-95-008, Institut d'Informatique, FUNDP, Namur, 1995.

[GACPRR89] González A., Alpuente M., Casamayor J.C., Pastor M.A., Ramírez M.J., Ramos I.; *PROTOLOG: A Conceptual Schema Facility for Automated Prototype Generation.* Proc.of the IASTED Intnl' Symposium on Applied Informatics, Switzerland, 1989: 43-46.

[G+83] Golshani F., et al.; *A Modal System of Algebras for Database Specification and Query/Update Language Support.* Procs. 9th. Intern.Conf. on VLDB, Florence, 1983.

[Harel84] Harel D.; *Dynamic Logic.* Handbook of Philosophical Logic II, Reider, 1985.

[Hoare85] Hoare C.A.R.; *Communicating Sequential Processes.* Prentice-Hall, 1985.

[LEG85] Lipeck U.W., Ehrich H.D., Gogolla M.; *Specifying Admissibility of Dynamic Databases Behaviour Using Temporal Logic.* Information Systems: Theoretical and Formal Aspects, North-Holland, IFIP, 1985.

[Davison87] Davison A. Polka: *A Parlog Object Oriented Language.* Dept.of Computing, Imperial College, London, 1988

[Pastor92] Pastor O.; *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el Modelo Orientado a Objetos.* PhD Thesis, DSIC-UPV, Valencia 1992.

[Ramos90] Ramos I.; *Logic and Object Oriented Data Bases: a Declarative Approach.* Proceedings of DEXA, 1990.

[RPCD93] Ramos I., Pastor O., J.Cuevas, J.Devesa; *Objects as Observable Processes.* Proc.of 4th Workshop on Deductive Approach to Information System and DB, Lloret de Mar, 1993.

[PR95] Pastor O., Ramos I.; OASIS 2.1.1.: *A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach.* SPUPV-95.788, Valencia 1995.

[SFSE87] Sernadas A., Fiadeiro J., Sernadas C., Erich H.-D.; *Abstract Object Types: A Temporal Perspective.* Proc.of Colloq. on Temporal Logic ans Specification, Manchester, 1987.

[SSG+91] Sernadas A., Sernadas C., Gouveia P., Resende P., Gouveia J.; *OBLOG-Object_Oriented Logic: An Informal Introduction.* Internal Report, INESC, Lisbon, 1991.

[Wieringa90] Wieringa R.J.; *Algebraic Foundations for Dynamic Conceptual Models.* PHD Thesis, Vrije Universiteit, Amsterdam, 1990.

[WM92] Wieringa R.J., Meyer J.J.Ch.; *Actors, Actions, and Initiative in Normative System Specification.* Annals of Mathematics and Artificial Intelligence, 1992.

Anexo

Se desarrolla la especificación completa del ejemplo mencionado anteriormente: el funcionamiento de un cajero automático, donde un cliente puede extraer o ingresar dinero.

```
#*********************************************
#DECLARACIÓN DE LA CLASE DE AGENTE CAJERO
#*********************************************
agent_class cajero
        identifier is  string
        agent_template  cajero_tp
agent_template          cajero_tp
vars
        c: cajero;
        cl: cliente;
        b: banco;
        t: tarjeta;
        p: string;
        o: operación;
        n: nat;
attributtes
        funciona: bool      derived from disponible < 10000
                            exported to cliente, banco;
        disponible: nat     default 1000000;
        tarj: tarjeta       exported to banco;
        pin: string         exported to banco;
        op: operacion       exported to banco;
        imp: nat            exported to banco;
        ok_trans: bool      derived from ok_tarj and ok_pin and
        ok_op and ok_saldo;
        ok_tarj: bool;
        ok_pin: bool;
        ok_op: bool;
        ok_saldo: bool;
actions
        solicita_tarj()     exported to cliente;
        solicita_pin()      exported to cliente;
        solicita_op()       exported to cliente;
        solicita_imp()      exported to cliente;
        solicita_din()      exported to cliente;
        escupe_tarj()       duration 1 second    exported to cliente;
        escupe_comp()       duration 2 seconds   exported to cliente;
        escupe_din()        duration 2 second    exported to cliente, banco;
        valida_tarj()       duration 1 second;
        valida_pin()        duration 1 second;
        valida_op()         duration 1 second;
        valida_saldo()      duration 1 second;
        valida_trans()      composed of valida_tarj()  valida_pin()
                                        valida_op()  valida_saldo()
                            exported to cliente;
        detiene_cajero()    exported to cliente, banco;
chek
        sometimes in the future disponible > 50000 since funciona = FALSE;
        always disponible >= 1000;
evaluate
        disponible = disponible - imp       when escupe_dinero() ends;
```

```
        disponible = disponible + imp      when cl.intro_dinero() ends;
        tarj = t                           when cl.intro_tarj(c,t) ends;
        pin = p                            when cl.intro_pin(c,p) ends;
        op = o                             when cl.intro_op(c,0) ends;
        imp = n                            when cl.intro_imp(c,n) ends;
        ok_tarj =TRUE   when valida_tarj() ends if b.operativa[tarj] = TRUE;
        ok_tarj =FALSE  when valida_tarj() ends          else;
        ok_pin =TRUE    when valida_pin() ends if b.pin[tarj] = pin;
        ok_pin =FALSE   when valida_pin() ends           else;
        ok_op=TRUE      when valida_op() ends  if b.permiso[tarj] = TRUE;
        ok_op=FALSE     when valida_op() ends            else;
        ok_saldo=TRUE   when valida_tarj() ends if op <> REINTEGRO
                                                or op = REINTEGRO
                                               and b.saldo[tarj] >= imp;
        ok_saldo = FALSE when valida_saldo() ends        else;
cause
        solicita_pin()      when cl.intro_tarj(c,t) ends;
        solicita_op()       when cl.intro_pin(c,p) ends;
        solicita_imp()      when cl.intro_op(c,o) ends
                            if op = REINTEGRO or op = INGRESO;
        valida_trans()      when cl.intro_op(c,o) ends       else;
        valida_trans()      when cl.intro_imp(c,n) ends;
        solicita_din()      when valida_trans() ends
                            if op = INGRESO and ok_trans = TRUE;
        escupe_tarj()       when valida_trans(c) ends        else;
        escupe_tarj()       when cl.intro_din(c) ends;
        escupe_comp()       when cl.retira_tarj(c) ends;
        escupe_din()        when cl.retira_comp(c) ends
                            if op = REINTEGRO and ok_trans = TRUE;
        solicita_tarj()     3 seconds after cl.retira_comp(c) ends     else;
        solicita_tarj()     3 seconds after cl.retira_din(c) ends;
force
        detiene_cajero()    when disponible < 10000 holds;
prevent
        solicita_tarj()     while funciona = FALSE holds;
ignore
        cl.intro_tarj(c,t)  when not solicita_tarj()
                            if c <> SELF or funciona = FALSE;
        cl.intro_pin(c,p)   when not solicita_pin()
                            if c <> SELF or funciona = FALSE;
        cl.intro_op(c,o)    when not solicita_op()
                            if c <> SELF or funciona = FALSE;
        cl.intro_imp(c,n)   when not solicita_imp()
                            if c <> SELF or funciona = FALSE;
        cl.intro_din(c,t)   when not solicita_din()
                            if c <> SELF or funciona = FALSE;
        cl.retira_tarj(c,t) when not escupe_tarj()
                            if c <> SELF or funciona = FALSE;
        cl.retira_comp(c,t) when not escupe_comp()
                            if c <> SELF or funciona = FALSE;
        cl.retira_din(c,t)  when not escupe_din()
                            if c <> SELF or funciona = FALSE;
#*************************************************
#DECLARACIÓN DE LA CLASE DE AGENTE CLIENTE
#*************************************************
agent_class cliente
        identifier is  string
        agent_template  cliente_tp
agent_template          cliente_tp
vars
        c: cajero;
        t: tarjeta;
        p: string;
        o: operación;
        n: nat;
actions
        intro_tarj(c,t)     duration 1 second    exported to cajero;
        intro_pin(c,p)      duration 2 seconds   exported to cajero;
        intro_op(c,o)       duration 1 second    exported to cajero;
        intro_imp(c,n)      duration 2 seconds   exported to cajero;
        intro_din(c)        duration 1 second    exported to cajero, banco;
        retira_tarj(c)      exported to cajero;
        retira_comp(c)      exported to cajero;
        retira_din(c)       exported to cajero;
cause
        intro_tarj(c,t)     when c.solicita_tarj() begins;
        intro_pin(c,p)      when c.solicita_pin() begins;
        intro_op(c,o)       when c.solicita_op() begins;
        intro_imp(c,n)      when c.solicita_imp() begins;
        intro_din(c)        when c.solicita_din() begins;
        retira_tarj(c)      when c.escupe_tarj() begins;
        retira_comp(c)      when c.escupe_comp() begins;
        retira_din(c)       when c.escupe_din() begins;
prevent
        retira_tarj(c)      when not c.escupe_tarj();
        retira_comp(c)      when not c.escupe_comp();
        retira_din(c)       when not c.escupe_din();
#*************************************************
#DECLARACIÓN DE LA CLASE DE AGENTE BANCO
#*************************************************
agent_class banco
        identifier is  string
        agent_template  banco_tp
agent_template          banco_tp
vars
        c: cajero;
        cl: cliente;
        t: tarjeta;
        p: string;
        o: operación;
        n: nat;
attributtes
        accesible: bool                                  default TRUE;
        operativa: tabla[tarjeta] de bool                exported to cajero;
        pin: tabla[tarjeta] de string                    exported to cajero;
        permiso: tabla[tarjeta,operacion] de bool        exported to cajero;
        saldo: tabla[tarjeta] de int                     exported to cajero;
actions
        conectar_red();
        desconectar_red();
        artivar_tarj(t);
        cancelar_tarj(t);
        asignar_pin(t,p);
        conceder_perm(t,o);
        revocar_perm(t,o);
        incr_saldo(t,n);
        decr_saldo(t,n);
        anular_cuenta(t)  composed of revocar_perm(t,REINTEGRO)
                                      revocar_perm(t,INGRESO) ;
                                     cancelar_tarj(t)
                          duration 5 minutes;
check
        sometimes in the future accesible = TRUE since accesible = FALSE;
        always pin[t] <> UNDEF;
evaluate
        accesible = TRUE                 when conectar_red() ends;
        accesible = FALSE                when desconectar_red() begins;
        operativa[t] = TRUE              when activar_tarj(t) ends;
        operativa[t] = FALSE             when cancelar_tarj(t) begins;
        pin[t] = p                       when asignar_pin(t,p) ends;
        permiso[t,o] = TRUE              when conceder_perm(t,o) ends;
        permiso[t,o] = FALSE             when revocar_perm(t,o) begins;
        saldo[t] = saldo[t] + n          when incr_saldo(t,n) ends;
        saldo[t] = saldo[t] - n          when decr_saldo(t,n) ends;
cause
        incr_saldo(c.tarj,c.imp)         when cl.intro_din(c) ends;
        decr_saldo(c.tarj,c.imp)         when c.escupe_din() ends;
force
        revocar_perm(t,REINTEGRO)        1 day after saldo[t] <= 0 holds;
prevent
        activar_tarj(t)     while saldo[t] < 0 holds;
        incr_saldo(t,n)     while permiso[t,INGRESO] = FALSE holds;
        decr_saldo(t,n)     while permiso[t,REINTEGRO] = FALSE holds;
ignore
        cl.intro_din(c)     while accesible = FALSE holds;
        c.escupe_din()      while accesible = FALSE holds;
hide
        operativa           to cajero        while accesible = FALSE holds;
        pin                 to cajero        while accesible = FALSE holds;
        permiso             to cajero        while accesible = FALSE holds;
        saldo               to cajero        while accesible = FALSE holds;
#*************************************************
#       FINAL DE LAS DECLARACIONES
#*************************************************
```

Ingeniería del Software

Experiencia práctica de implantación y uso de MÉTRICA V.2 en una empresa de ingeniería y desarrollo de S.I.

Juan Peña Amaro,
Consultor Senior de SADIEL, S.A.
Jesús Macías Castellano,
Director Técnico de SADIEL,S.A.

1. Introducción

Se presenta en este artículo la experiencia práctica de adopción de MÉTRICA v.2 como metodología de trabajo en una PYME (Pequeña y/o Mediana Empresa) dedicada a la ingeniería y desarrollo de sistemas de información, con detalles referentes a su implantación y su adaptación a entornos de desarrollo concretos, así como los resultados y conclusiones más relevantes que se han obtenido.

Desde hacía algunos años se disponía de un método propio, no excesivamente formalizado, para el desarrollo de proyectos y el personal se encontraba familiarizado con el uso de métodos y técnicas propias de ingeniería de software. También ya se venían utilizando diversas herramientas CASE (Oracle*Case, Excelerator II, Easy-Case, Case NEW/Predict), aunque básicamente como 'upper-case', para las fases de análisis y diseño.

Antes de la aparición de Métrica v.2, en marzo de 1993, se había aplicado Métrica v.1 en algunos proyectos de desarrollo, pese a conocer las limitaciones de esta primera versión; en estos casos se valoró más la experiencia formativa que los beneficios reales obtenidos.

2. Circunstancias que motivaron la toma de decisión

Entre las circunstancias que influyeron en la toma de decisión de la adopción de Métrica v.2 como metodología de desarrollo figuran las que se comentan a continuación.

- La Administración Autonómica Andaluza, nuestro principal cliente, comenzó a recomendar y, más tarde, a exigir el uso de Métrica v.2 en los pliegos de prescripciones técnicas de sus concursos de desarrollo de proyectos informáticos.

- Métrica v.2 se anunciaba como un marco metodológico de referencia común a las diversas administraciones públicas del Estado Español y con posibilidades de aplicarse en el futuro a otros segmentos del sector, públicos y privados.

- La adopción de Métrica v.2 podría suponer la eliminación de los recursos y costes asignados a mantener y adecuar los métodos propios: la evolución de Métrica deberá soportarse por su propietario, el Ministerio para las Administraciones Públicas.

- La utilización de métodos basados en estándares y contrastados públicamente contribuiría a mejorar nuestro Sistema de Calidad, lo que nos ayudaría en nuestro empeño de obtención del certificado de empresa en relación a las normas ISO-9000.

3. Objetivos planteados en la fase de implantación

Para la implantación de Métrica v.2 se creó expresamente un equipo de trabajo, al que se le fijaron una serie de objetivos:

- Elaboración de unas Guías Metodológicas específicas, resultado de la adaptación de Métrica v.2 a los entornos de desarrollo más usuales en la empresa.

- Liderazgo y fomento de la utilización práctica de dichas guías de forma inmediata por el resto del personal en proyectos reales de desarrollo, mediante la integración y empleo de las herramientas Case apropiadas en cada caso.

- Elaboración del material didáctico que permitiera la formación y facilitara la posterior realización de proyectos bajo este marco metodológico, minimizando el impacto que pudiera tener el cambio cultural, organizativo y procedimental, con especial atención a lo que pudiera afectar al personal con menor experiencia en el uso de métodos.

- Propuesta de actuaciones para la mejora de la calidad en los productos y en el proceso de desarrollo, tomando como referencia el ***'Plan General de Garantía de Calidad aplicable al desarrollo de equipos lógicos (PGGC)'*** del Ministerio para las Administraciones Públicas.

4. Planificación del proceso de implantación

Métrica v.2 establece en el capítulo 1.6 de su Guía de Referencia los criterios de referencia para el proceso de su implantación. En particular en el caso que nos ocupa, se planificó y realizó según refleja el gráfico de la **figura 1,** cuyos pasos se detallan.

PASO 1: Seleccionar entornos de desarrollo

Objetivo: Identificación de los entornos tecnológicos de desarrollo (plataformas micro, mini y mainframe) más usuales en la empresa y selección de los de mayor interés para la implantación.

PASO 2: Seleccionar herramientas CASE

Objetivo: Identificación de las herramientas Case, de entre las conocidas y disponibles, las que tenían mayor utilización en cada uno de los entornos de desarrollo seleccionados.

PASO 3: Elaborar el plan de implantación para cada entorno seleccionado

Objetivo: Elaboración de un plan detallado de adaptación e implantación para cada entorno seleccionado, con recursos involucrados y plazos. Una de las conclusiones de este plan fue la conveniencia de reducir el número de entornos seleccionados inicialmente y dejar los restantes para una fase posterior.

Se constituyó un equipo de trabajo compuesto por jefes de proyecto y consultores con experiencia en los entornos seleccionados y en dimensionamiento o estimación de proyectos, que asumió la responsabilidad de elaborar el plan, así como supervisar la ejecución de los restantes pasos y obtener conclusiones en base a debate común sobre los resultados de los siguientes pasos.

La Dirección de la empresa, impulsora desde el primer momento de este proceso, recibía de este equipo de trabajo información sobre la evolución de los trabajos.

PASO 4: Adaptación de Métrica v.2 a los entornos y herramientas CASE seleccionados

Objetivo: Elaboración de una primera versión de la GUÍA ADAPTADA para recoger los trabajos de adaptación de Métrica v.2 a las características específicas de la empresa y a cada uno de los entornos de desarrollo seleccionados. Estos trabajos se concretaron en las siguientes actividades:

- Definir el mapa de actividades de Métrica en cada caso
- Asignar funciones y responsabilidades
- En las fases de mayor dependencia con el entorno tecnológico, definir normativas, estándares y criterios de nomenclatura y normalización en uso o aportadas por el propio entorno.
- Definir los productos resultados de cada actividad.
- Elaborar un informe de adecuación a Métrica v.2 para las herramientas CASE y de desarrollo elegidas.
- Establecer los procedimientos e instrumentos de control de calidad y elementos auxiliares de garantía de calidad.

PASO 5: Formación

Objetivo: Preparar en Métrica v.2 y en las GUIAS ADAPTADAS al personal técnico implicado en el desarrollo de proyectos.

En una primera fase la formación fue impartida de forma particular y específica al equipo de trabajo asignado a cada proyecto nuevo y como fase previa obligada antes de abordar su comienzo.

Más tarde se generalizó, con un plan de formación adecuado, al resto del personal técnico involucrado en desarrollo de proyectos.

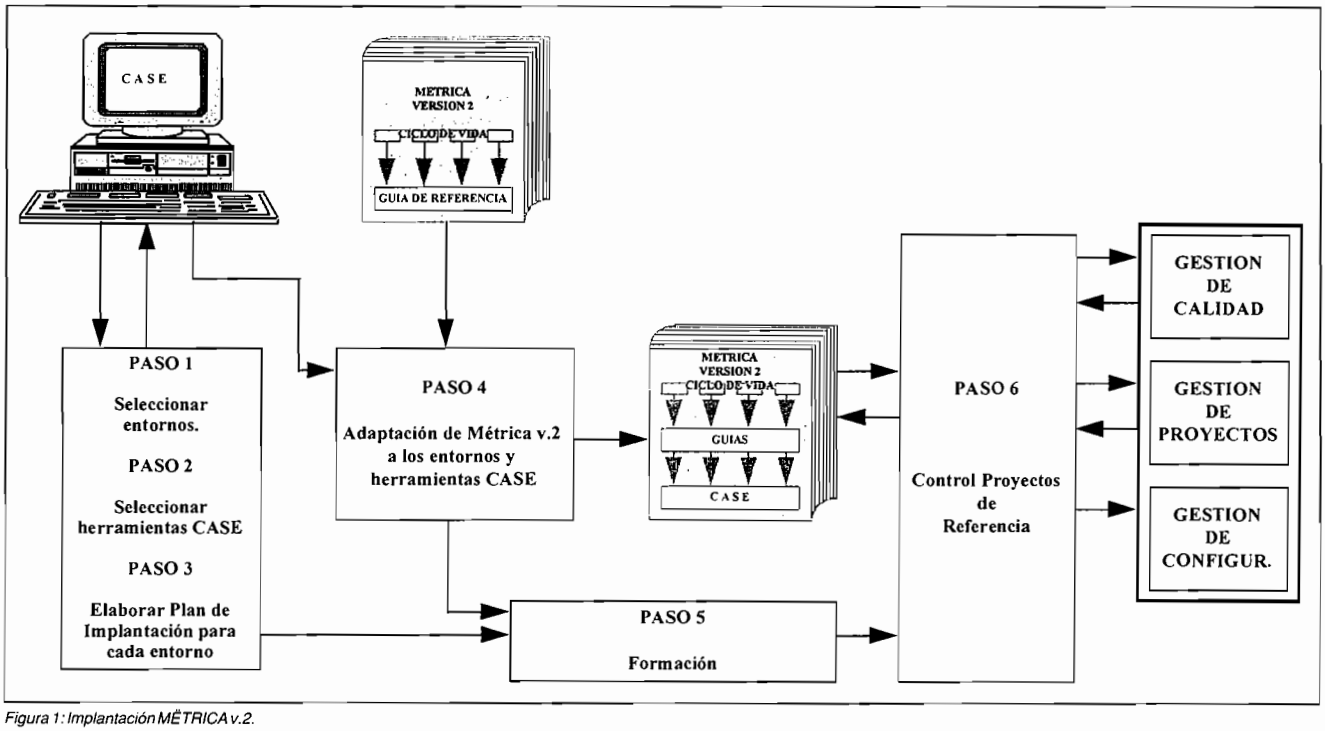


Figura 1: Implantación MÉTRICA v.2.

Así mismo, y a demanda de uno de nuestros clientes, se elaboró e impartió (se sigue impar-tiendo) un programa completo de formación en Métrica v.2 y técnicas asociadas de contenidos eminentemente prácticos.

PASO 6: Control de proyectos de referencia

Objetivo: Contrastar y comprobar la idoneidad de la adaptación realizada en proyectos en curso o próximos a realizar en cada uno de los entornos seleccionados y, en su caso, modificar o implementar las sugerencias de mejora. Para ello se asignó a un miembro del equipo de trabajo funciones de seguimiento y control de la adaptación y de asesoramiento y soporte en estos proyectos de referencia.

5. Algunos datos sobre el proceso realizado

Métrica v.2 indica que un proceso de adaptación debe durar un período de tiempo del orden de uno a dos meses.

Sin embargo esta referencia no es aplicable en el caso que nos ocupa, si se tiene en cuenta que la decisión formal de adopción de la metodología fue el resultado de un proceso de maduración y convencimiento paulatino y que gran parte de las actividades de adaptación e implantación fueron realizadas por los técnicos con carácter marginal y al mismo tiempo que debían atender otros trabajos en curso.

A título orientativo en la **figura 2** se presenta un gráfico con indicación de los plazos y recursos consumidos en el proceso de implantación.

6. Resultados y conclusiones

Como resultado del proceso de adaptación de Métrica v.2, se obtuvo para cada entorno tecnológico de desarrollo seleccionado una GUÍA ADAPTADA, una de cuyas páginas muestra como ejemplo la **figura 3**. Esta Guía es una versión de la de Métrica v.2 totalmente adaptada al entorno tecnológico y a experiencias anteriores existentes.

Así mismo como resultado del proceso, hoy se puede asegurar que:

- Se dispone de una metodología totalmente ajustada, con un gran nivel de detalle y practicidad.
- Se asegura la utilización y seguimiento de la metodología y de las herramientas apropiadas.
- Se facilitan las acciones y realización del control de calidad de los proyectos y productos.
- En cada caso se clarifica los mínimos metodológicos a verificar, exista o no compromiso de metodología, sin incurrir en excesos.
- El proceso de adaptación no ha modificado la estructura fundamental de la metodología (organización, fases, módulos, actividades, técnicas y productos finales).

Como conclusiones de interés, en primer lugar conviene destacar la necesidad de una formación en Métrica v.2 para el personal informático. Esta también debe contemplar el personal usuario en lo que se refiere a destacar las funciones y responsabilidades en las fases en que más participan, a conocer el ciclo de vida y los productos y a familiarizarse con las técnicas de diagramas de flujo de datos y modelos de datos.

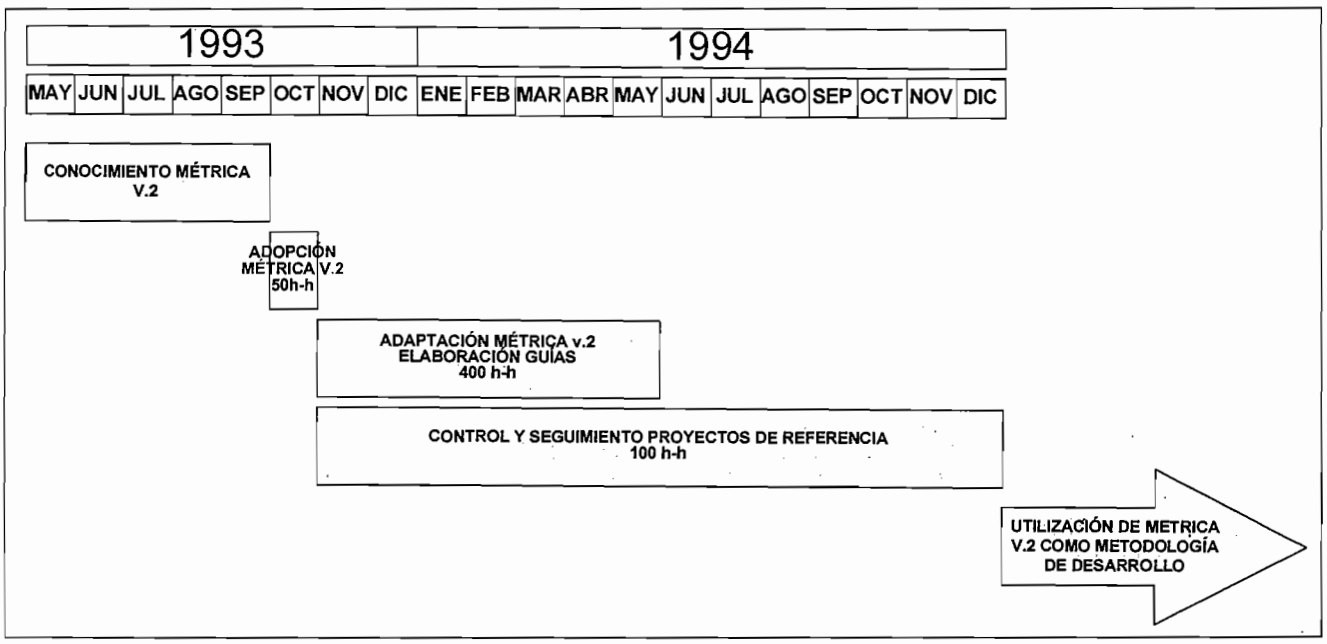


Figura 2: Calendario del proceso de Implantación

En segundo lugar la necesidad, a la hora de aplicar a un proyecto concreto la GUÍA ADAPTADA correspondiente a su entorno, de distinguir en cual de los dos supuestos siguientes se encuentra:

- ***el cliente exige la aplicación de la metodología Métrica.*** En este caso se deben realizar todas las actividades especificadas en la guía y con este supuesto debe estimarse y abordarse el proyecto;
- ***el cliente no exige Métrica pero se decide emplearla.*** En las guías figuran, claramente diferenciadas, dos clases de actividades: las que deben realizarse obligatoriamente para alcanzar un mínimo metodológico y las que pueden o no realizarse opcionalmete, en función de las características, circunstancias y limitaciones económicas del proyecto.

Hemos podido constatar cómo la plena y rigurosa aplicación de Métrica v.2 supone un incremento importante de plazos y recursos dedicados, y en definitiva de los costes de desarrollo, incremento que según nuestra experiencia puede estar entre un rango obtenido de multiplicar por 1,7 en proyectos medios y por 2,5 en proyectos pequeños los plazos y recursos correspondientes a desarrollos realizados siguiendo nuestro método anterior. Aunque no se dispone de datos de referencia, es de suponer que los efectos positivos de la aplicación metodológica se observarán en los futuros costes de mantenimiento. Es importante que los responsables de la estimación y evaluación de proyectos, en los organismos públicos y empresas privadas, tengan en cuenta los sobre costes que puede llegar a suponer la aplicación de ésta u otra metodología.

Desde la finalización de este proceso de implantación es práctica habitual en nuestra empresa la utilización de Métrica v.2 como metodología de desarrollo por parte de todo el personal técnico involucrado en el desarrollo de proyectos. A partir de las experiencias de este uso habitual se mantienen actualizadas las Guías Adaptadas.

Finalmente, y como continuación de este proceso, hemos abordado un paso más en el nivel de madurez como empresa dedicada a la ingeniería y desarrollo de software, atacando simultáneamente dos frentes. Por un lado hemos elaborado un modelo de estimación y evaluación de recursos y plazos adaptado a las actividades de Métrica v.2. La herramienta obtenida con el desarrollo informático de este modelo está suficientemente probada y permite obtener una mejor y más fiable estimación y evaluación inicial de los proyectos, así como su actualización a medida que avanza el desarrollo. Por otra parte hemos establecido mecanismos de medida y control de recursos consumidos y plazos en actividades y proyectos, cuyo análisis permite tomar acciones para optimizar el uso de los recursos y analizar y corregir las causas de las desviaciones, y al mismo tiempo es posible alimentar con información real el modelo de estimación antes citado.

ACTIVIDAD | O | DTS 1 | DISEÑAR LA ARQUITECTURA FÍSICA DEL SISTEMA

TAREA	DESCRIPCION Y OBJETIVOS
1.1 Diseño de la estructura modular del Sistema	-Definir el conjunto inicial de actividades físicas del Sistema tomando como punto de partida los DFD construidos en EFS 1, teniendo en cuenta las divisiones que presentan y las particularidades de los procesos que describen; tipo de tratamiento, elaboración de informes, realización de algoritmos específicos,... -Asegurar que todas las funciones realizadas por el Sistema quedan soportadas por las actividades que se han descrito. -Diseñar la estructura modular del sistema mediante Diagramas de Estructura de Cuadros (Constantine). Se precisan al menos: . Estructura de alto nivel que presente el Sistema descompuesto en los subsistemas que lo componen (partir del DFD de Nivel 1 de EFS 1). . Un conjunto de Diagremas de Estructura de Cuadros para cada une de las actividades identificadas al principio de esta tarea (partir de los DFD de nivel inferior de EFS 1). Refinar los diagremas añadiendo; módulos que detallen el tratamiento de las actividades físicas; símbolos que detallen aspectos del tratamiento (iteraciones, opcionalidad, .)
1.2 Descripción de interfases entre módulos del Sistema.	-Definir la información (tanto de control como datos de la aplicación) comunicada entre módulos; su formato, origen y destino.
1.3 Descripción de interfases con otros sistemas.	-Definir posibles interfases con otros sistemas existentes o por desarrollar.
1.4 Descripción de interfases de usuario	-Establecer el diseño detallado de las interfases de usuario referido al entorno técnico en que trabajarán dichas interfases, considerando la estructura modular que se ha definido (consultar los resultados de EFS 4.1 y revisar diálogos, atributos de pantalla, etc.)
1.5 Definición de componentes del Sistema	-Partiendo de la descripción de funciones de EFS, definir las unidades básicas del desarrollo. En principio cada módulo identificado constituirá un programa aunque podrán agruparse con fines de optimización -Producir una especificación detallada de cada componente a desarrollar; lógica de tratamiento, llamadas a componentes, secuencia de paso de parámetros,... -Definir el pseudo-código incluyendo; E/S, accesos a BD, llamadas, estructura de control interno, interfases con otros sistemas.

PARTICIPANTES	FUNCIONES Y RESPONSABILIDADES	ESTIMACION RECURSOS
Especialistas en sistemas	Prestar asistencia técnica	
EDS (AS)	Ejecutar las tareas de la Actividad	3 jornadas a 2,5 semanas
EDS (Analista Programador -AP-)	Ejecutar las tareas de la Actividad	4 a 20 semanas

TECNICA	HERRAMIENTAS	COMENTARIOS
1.1-1.3, 1.5 Diagrama de Estructura de Cuadros de Constantine	EasyCase: -Abrir, para cada diagrama, un nuevo fichero de tipo STC (Structure Chart) -Dentro de las opciones del diagrama (Chart Preferences), desactivar 'pipes'. -Tras identificar el diagrama, incluir en el todos los objetos que procedan (modulos, conexiones, signos de comunicación, secuencias, iteraciones, decisiones, almacenes, dispositivos, etc.), etiquetándolos y definiendolos en el DDE. -Si se estimese oportuno, se completarán los campos (del DDE) que siguen; • 'Alias' • 'Misc #1, #2, #3' • 'Short Description'	Se verificarán los diegrama STC mediante la opción 'Rule Check', eliminando los errores relevantes.
1.4 Prototipado de pantallas y diálogos		
1.5 Descripción de cada componente mediante pseudo-código, lenguaje estructurado, etc.	EasyCese; -Definir, para los componentes a describir, hijos de tipo Primitive Proccess Spec. (pps) o Elementary Proccess Desc. (epd) que permiten la introducción de la información.	La definición de los procesos se efectuará en forma de pseudo-código.

PRODUCTO	DOCUMENTO	COMENTARIOS
1.1 Actividades físicas del Sistema, diagramas de estructura de cuadros de alto y bajo nivel.	Apartado 1.1 del Documento de Diseño Técnico (DDT).	
1.2 Descripción detallada de interfases de control y datos entre módulos.	Apartado 1.1.1 del DDT.	
1.3 Descripción de interfases con otros sistemas.	Apartado 1.2 del DDT.	
1.4 Descripción detallada de las interfases de usuario	Apartados 1.2.2 y 1.3.3 del DDT.	
1.5 Definición y descripción de los componentes del Sistema	Apartado 1.3.1 del DDT.	

CONTROL CALIDAD

PROCEDIMIENTOS	INSTRUMENTOS	DOCUMENTACION
No aplica.		

Figura 3: Página ejemplo

Ingeniería del Software

Prototipado en MÉTRICA 2: ejecución de especificaciones basadas en Diagramas de Flujo de Datos

Manuel A. González Rodríguez, Ambrosio Toval Álvarez, Jesús García Molina
Grupo de Ingeniería de Software, miembro de RENOIR (European Engineering Network of Excellence)
Dpto. Informática y Sistemas, Universidad de Murcia

e-mail: {manolo | atoval | molina}@dif.um.es;
tlf: 34-68-307100 ext. 2050; fax: 34-68-364151

Resumen: *la deseada implantación de las metodologías de desarrollo de Sistemas de Información (SI) se encontraría seriamente limitada sin herramientas que ayuden a los productores de software a completar las guías propuestas. Este artículo presenta una aportación original para generar y ejecutar prototipos a partir de especificaciones estructuradas de acuerdo con los criterios de la metodología oficial MÉTRICA 2: utilización de técnicas estructuradas formales/rigurosas y utilización de prototipos durante la especificación funcional del sistema. Presentamos un lenguaje formal, a la vez que simple, denominado MSL (Mini-Specifications Language), para escribir las miniespecificaciones (ME) de los procesos reflejados en los sucesivos Diagramas de Flujo de Datos (DFD) del sistema. El proyecto se aborda íntegramente desde una perspectiva rigurosa que incluye una formalización de técnicas estruc-turadas de MÉTRICA 2 en una lógica de primer orden. La ejecución visual usando los mismos DFD posibilita que estos aspectos formales no sean visibles a los usuarios. Existe ya una primera versión experimental de esta herramienta en entornos PC.*
Palabras clave: *MÉTRICA 2, Especificaciones Formales, Prototipado, Técnicas Estructuradas, Ingeniería de Requisitos, Programación Lógica.*

1. Introducción

La Metodología de Desarrollo de Sistemas en la Administración Española (MÉTRICA 2) [MAP 95] se ha implantado con el fin de solucionar la problemática que resulta de la escasa documentación de los sistemas y de la falta de comunicación con los usuarios durante el proceso de desarrollo, lo que genera productos que no responden totalmente a las necesidades de los usuarios. En este sentido, define un conjunto de métodos, procedimientos, técnicas y herramientas que facilitan la construcción de SI. El marco de trabajo de MÉTRICA 2 se estructura en fases que a su vez se dividen en módulos. Nuestro trabajo realiza su contribución básicamente en la fase de *'Análisis de Sistemas'*. Por un lado, formalizando determinadas técnicas estructuradas utilizadas en MÉTRICA 2; y por otro lado, aportando una herramienta conversacional y visual para el prototipado funcional que es posible utilizar en los módulos *'ARS. Análisis de Requisitos del Sistema'* y *'EFS. Especificación Funcional del Sistema'*, con el fin de generar y ejecutar prototipos a partir de las especificaciones, y validar/verificar los requisitos del sistema junto a los usuarios. También, en otras fases de MÉTRICA 2 es útil esta contribución en la medida en que se deban mantener, validar y/o verificar requisitos con los usuarios.

Autores reconocidos como [Harel 92] [Fuchs 92] [Goguen 94] defienden la idea de que para tener éxito en el desarrollo de software es necesaria una adecuada planificación y ejecución de las fases de especificación y diseño que incluya la utilización de prototipos. Pero desgraciadamente, no existen herramientas que permitan generar y ejecutar prototipos funcionales. Por ejemplo, MÉTRICA 2 consciente de que la participación activa del usuario es una condición imprescindible para el éxito en el desarrollo de SI, propone técnicas interactivas, entre ellas el uso de prototipos, pero no menciona en ningún momento una herramienta o método para la construcción de prototipos funcionales (sólo se refiere a prototipos de interfaz: pantallas, diálogos, formularios e informes).

El trabajo realizado se podría trasladar, sin grandes cambios, a otros métodos que también utilicen técnicas estructuradas, como p. ej. las metodologías oficiales SSADM (Inglaterra), MERISE (Francia) que han supuesto un verdadero cambio socio-técnico en el desarrollo de SI en sus respectivos países. Aún más, algunas de las metodologías y herramientas CASE Orientadas a Objetos (OO) también utilizan estas técnicas, por ejemplo OMT (Object Modelling Technique) [Rumbaugh 91].

MÉTRICA 2 recomienda, al menos en su versión actual, el uso de técnicas estructuradas, entre ellas los DFD. Estas técnicas resultan muy adecuadas para describir requisitos funcionales de sistemas complejos: su facilidad de aprendizaje y uso las convierte en un buen método de comunicación entre los usuarios y los productores de software. Sin embargo, carecen de un fondo teórico sólido, lo que dificulta los procedimientos de validación y verificación automática. Otra importante desventaja es la distancia ('gap' semántico) que existe entre la última especificación (*'DTS. Diseño Técnico del Sistema'* en MÉTRICA 2) y la fase de construcción de sistemas, lo que dificulta la obtención de prototipos funcionales a partir de las especificaciones. Los métodos formales (Z, VDM, especificaciones algebraicas [Ehrig et al 92]) ofrecen una alternativa válida al proceso de desarrollo del software; en realidad, constituyen una de las tendencias que más acerca a la Ingeniería del Software a una verdadera disciplina científica. Aun así, nosotros estamos convencidos que métodos informales como los estructurados todavía desempeñan, y desempeñarán en los próximos años, un papel muy importante. Dada la dificultad que están encontrando los métodos formales para ser aceptados por la comunidad de desarrollo de SI, resulta interesante intentar formalizar metodologías intuitivas y ampliamente consolidadas, como es el caso del Análisis y Diseño Estructurado, combinando una aproximación rigurosa al proceso de desarrollo con metodologías no formales. En esta línea, han aparecido recientemente diversas formalizaciones de algunos de los conceptos de las técnicas estructuradas, bien desde un punto de vista algebraico [Tse 91] [Tao et al. 91] [France et al. 89], bien desde un punto de vista lógico [Tse et al 94], o mediante VDM [Larsen et al 94]; en [Fuggeta et al 93] se emplea un diagrama similar a una red de Petri junto con un lenguaje estructurado. Los trabajos citados ponen el énfasis en la traducción de las especificaciones [Tse 91], en la combinación de DFD y diagramas de Entidad-Relación [Fuggeta et al 93] y en los diagramas de estructura [Tse et al. 94], etc... Nosotros nos centramos en la formalización de los DFD, en la descripción formal de la lógica

de los procesos utilizando un lenguaje a la vez fácil y expresivo, denominado MSL, y en la generación automática de prototipos ejecutables. Nuestro grupo de investigación adopta una estrategia similar en la formalización del proceso de especificación de requisitos desde un punto de vista OO [Toval et al 94][OOAP 95].

Este trabajo se sitúa en la fase de Ingeniería de Requisitos, en particular, hace posible la ejecución de los DFD. Incluye un lenguaje de miniespecificaciones (MSL) para escribir los requisitos lógicos de los procesos. Incorpora el rigor de los métodos formales (concretamente lógica de primer orden) lo que supone una contribución a la obtención de software de calidad, con requisitos validados y verificados, no sólo de pantallas o formularios sino, lo que es más importante, sobre el comportamiento y la funcionalidad del sistema [Fuchs 92].

La posibilidad de escribir ME y, a partir de ellas y los DFD, de generar prototipos, produce un efecto sinérgico que incrementa la productividad, ya que se permite en la fase de análisis la validación por parte de los usuarios de los productos obtenidos, tal como se recomienda en MÉTRICA 2.

Es importante resaltar que la herramienta que proponemos es integrable en un entorno CASE estructurado, lo que conlleva una serie de ventajas relativas al uso de un vocabulario común consolidado como son los DFD, de los cuales mantenemos la notación original.

En el resto de esta sección se describe el proceso de prototipado y su aplicación en MÉTRICA 2. En la sección 2 se describen las funciones del entorno de prototipado. La sección 3 se dedica a pormenorizar el lenguaje de miniespecificaciones MSL. En la sección 4 se describe, desde un punto de vista más tecnológico, el proceso automático de obtención de prototipos a partir de las especificaciones. Finalmente, en la sección 5 se resumen las conclusiones y las líneas de investigación abiertas. La sección 4 puede obviarse por los lectores que sólo sean usuarios de la metodología MÉTRICA 2.

1.1. MÉTRICA 2 y el prototipado

Atendiendo a [Tanik et al. 89], el diseño de prototipos consiste en construir una versión reducida del sistema que sirva como base en la construcción del sistema final. La ejecución de prototipos ayuda a que los productores de software y los usuarios validen los sucesivos modelos de un sistema con el fin de comprobar que su comportamiento cubre las necesidades de los usuarios [Luqi 89]. La **figura 1** describe todo el proceso de prototipado. MÉTRICA 2 propone la utilización de técnicas interactivas que permitan al usuario familiarizarse con el sistema y colaborar en su desarrollo, para facilitar la participación de los usuarios en el desarrollo de los SI, hecho que considera imprescindible para que los requisitos identificados se ajusten a las necesidades de los usuarios. Con el uso de prototipos el proceso de desarrollo se mejora substancialmente, ya que los cambios en la especificación en las fases iniciales del desarrollo se reflejan en el prototipo funcional en vez de en el programa final, con el consiguiente ahorro de esfuerzo y aumento de productividad. Además, la comprobación y la traducción automática de los sucesivos modelos asegura la consistencia y la completitud de las especificaciones.

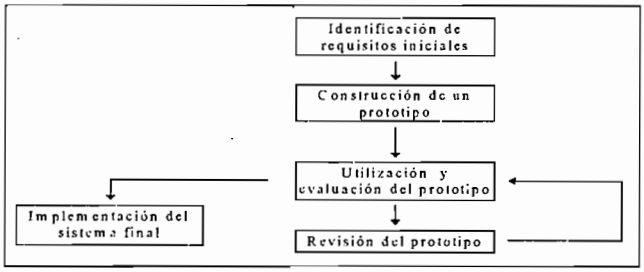


Figura 1: El proceso de prototipado

Tanto en MÉTRICA 2 como en la gran de la mayoría de las herramientas CASE que integran metodologías estructuradas, las funciones de prototipado sólo consisten en la generación automática de pantallas y en herramientas que simulan el aspecto visual de los informes o formularios. MÉTRICA 2 menciona la importancia de los prototipos funcionales pero sólo incluye en las guías metodológicas el uso de prototipos de interfaz. Aunque este tipo de prototipos ayudan a la hora de validar los requisitos del usuario, no se pueden considerar suficientes pues no cubren la parte esencial de los requisitos de un SI: los requisitos funcionales y de comportamiento.

En la **figura 2** se muestra una visión general de la metodología MÉTRICA 2 donde aparecen sombreadas las actividades que consideramos que más pueden aprovechar las ventajas de la utilización de prototipos funcionales. El prototipado funcional se centra en la fase de análisis, cuyo fin es capturar los requisitos del sistema (módulo ARS) e identificar la arquitectura lógica del nuevo sistema (módulo EFS).

En concreto, consideramos que la técnica del prototipado funcional se puede incluir en las actividades:

ARS.3. Diseñar el Modelo Lógico Actual. La ejecución de los DFD que se generan en esta actividad es útil para que analistas y usuarios comprueben propiedades del Modelo aprehendido.

ARS.4. Estudiar alternativas de construcción. Las distintas alternativas se representan mediante un modelo lógico de procesos que describe sus rasgos más distintivos. En la elección de una de las alternativas puede desempeñar un papel clave la utilización de prototipos que permitan identificar las ventajas e inconvenientes de cada opción.

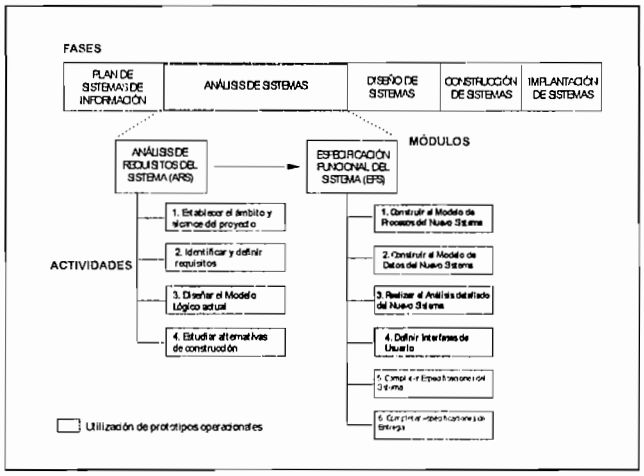


Figura 2: Metodología METRICA 2 y papel del prototipado funcional

EFS.1. Construir el Modelo de Procesos del Nuevo Sistema. Al igual que en ARS.3 los prototipos funcionales son muy útiles como herramienta conversacional en el diálogo de los analistas con los usuarios para conocer sus necesidades y para la descripción en detalle de las especificaciones funcionales del sistema mediante especificaciones formales ejecutables para obtener la validación/verificación de los requisitos.

EFS.3. Realizar el Análisis detallado del Nuevo Sistema. En esta actividad que es un refinamiento de la EFS.1 son válidos los mismos criterios.

EFS.4. Definir Interfaces de Usuario. MÉTRICA 2 sólo incluye en esta actividad el uso de prototipos de interfaz y, aunque resalta una vez más la importancia de los prototipos funcionales, no los incorpora a las guías metodológicas. El uso de prototipos que simulen el comportamiento del sistema con datos específicos es posible con la herramienta propuesta y es un complemento ideal a los prototipos de interfaz.

Nuestra aportación intenta llenar el vacío de herramientas de prototipado funcional y va más allá. Al utilizar la interfaz gráfica de los DFD para su ejecución, evita de esta manera que el usuario no experto interactúe directamente con el formalismo subyacente (en nuestro caso lógica de primer orden) y permite la definición de ME en un lenguaje muy próximo al lenguaje estructurado tradicional.

2. Descripción de un entorno de prototipado

Este trabajo está inspirado en el trabajo de [Goble 89], donde se relata cómo escribir de forma manual especificaciones en PROLOG (muy cercanas a la máquina) para construir un prototipo. Sin embargo, desde nuestro punto de vista, esta propuesta tiene al menos dos inconvenientes: primero, se pierden las ventajas de poseer una especificación estructurada clara y comprensible; segundo, las miniespecificaciones de los procesos pierden portabilidad al ser dependientes del lenguaje PROLOG. En el lado contrario a la aproximación de Goble, se encuentra el enfoque tradicional que consiste en escribir especificaciones en un Lenguaje Natural Estructurado (muy cercanas al hombre), un lenguaje que no es formal pero que es flexible y claro para los usuarios, aunque esta flexibilidad e 'informalidad' dificulta su compilación, la generación de prototipos, verificación automática de corrección, etc...

[Molina et al. 92] y [Fugetta et al 93] contienen dos propuestas para la ejecución de los DFD. En [Molina et al. 92] se describe SAPE, que sería la primera versión de un entorno de prototipado que integra MSL. En [Fugetta et al 93] se intenta solucionar la ambigüedad que presentan los DFD, introduciendo un lenguaje formal al que se incorporan características típicas de los lenguajes estructurados, como PASCAL, y una nueva notación para los DFD, que incluye información del diagrama de E-R. En nuestra opinión, esta propuesta se aleja de la notación original de los DFD ampliamente aceptada, lo que creemos un claro inconveniente al apartarse del estándar; además, el lenguaje que se utiliza es demasiado parecido a un lenguaje de programación con detalles de implementación.

Nosotros adoptamos una postura 'ecléctica' que nos sitúa entre la flexibilidad del Lenguaje Estructurado y el rigor de PROLOG. Proponemos un lenguaje estructurado imperativo, MSL, con algunas restricciones que lo convierten en un lenguaje formal, lo que permite la manipulación automática de las miniespecificaciones escritas en MSL. Nuestra filosofía apunta en tres direcciones que se pueden representar como las tres hojas de un trébol (**figura 3**). La hoja superior representa un CASE típico que proporcione soporte a técnicas estructuradas y que incluya un administrador de proyectos, un editor visual para editar (y posteriormente ejecutar) las especificaciones en términos de DFD, un editor de miniespecificaciones MSL y un Sistema de Soporte a la Ejecución (en nuestro caso y de forma transparente al usuario, un intérprete PROLOG).

La **figura 4** proporciona, mediante un DFD, una descripción simplificada de las funciones del entorno de prototipado. A partir de la información de los DFD y el Diccionario de Datos (DD) es posible conocer, para cada uno de los procesos, los flujos de entrada/salida, los almacenes de datos que utiliza, etc. En primer lugar, se convierte la representación gráfica de un DFD a 'hechos' PROLOG (es decir, a una representación formal en lógica de primer orden); a continuación, se genera de forma automática, para cada uno de los procesos, una plantilla de código MSL validado que incluye la descripción de los flujos de datos, los almacenes de datos e información relativa a los procesos, lo que supone ya una ayuda automática a la posterior especificación detallada de la funcionalidad de los procesos (el analista utiliza el lenguaje MSL para completar esta plantilla con los requisitos lógicos del programa). Finalmente, se realiza automáticamente un análisis de consistencia y se genera, internamente, un programa PROLOG equivalente a un prototipo funcional de la especificación.

La formalización en lógica de primer orden de los DFD y del código MSL permite la demostración formal de propiedades de

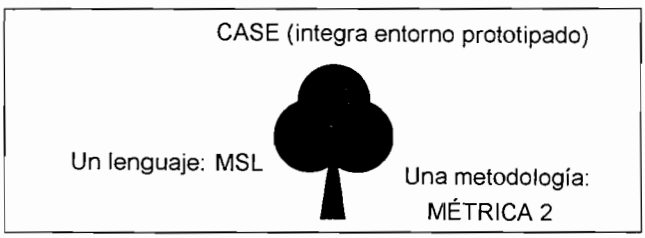


Figura 3: Entorno de prototipado

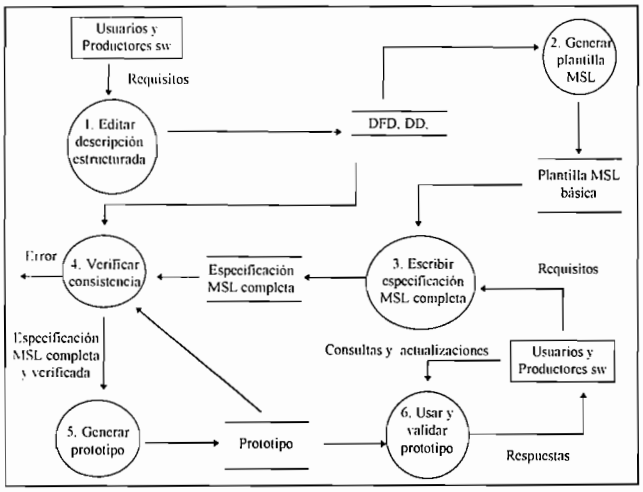


Figura 4: Descripción funcional de un entorno de prototipado

la especificación. Algunas de estas propiedades son meramente sintácticas, otras se refieren a la coherencia entre la especificación MSL completa y los DFD (p. ej., comprobar si todos los componentes de los DFD están correctamente descritos en la especificación MSL, si todos los flujos de entrada/salida aparecen en la lista de argumentos,...); otras propiedades hacen referencia a la corrección semántica de los DFD (p.ej., comprobar que desde cada uno de los procesos es posible acceder a una fuente y a un sumidero de datos, comprobar si un proceso explota, si los archivos se heredan de un nivel a otro,...).

Otra propiedad que es posible comprobar es la completitud de las miniespecificaciones MSL, p. ej. para detectar partes indefinidas. Además, esta formalización facilita posteriores traducciones del modelo, generación de código, etc. Productores de software y usuarios pueden utilizar el prototipo generado para verificar los requisitos funcionales del sistema de información en desarrollo utilizando la herramienta de forma conversacional. A partir de ese momento, es posible refinar la especificación (DFD, MSL), combinar prototipos funcionales con otro tipo de prototipos, 'simular' la ejecución de los procesos con datos de entrada proporcionados por los usuarios, etc.

Usando el prototipo equivalente a un ejemplo de gestión de pedidos (**figura 5**), se puede preguntar el valor de los flujos de datos, de los almacenes, de los procesos de entrada de datos o en general los resultados de la ejecución de cualquiera de los procesos descritos. Las respuestas permiten 'capturar' el comportamiento funcional del sistema en desarrollo. Si se desea conocer p.ej. todos los pedidos tasados y aceptados, sólo hay que picar con el ratón en el flujo etiquetado como 'pedido tasado aceptado': aparece un panel de diálogo, en el que se deben de rellenar (o dejar en blanco como variables libres) los parámetros del flujo de datos, entre ellos el nombre del cliente.

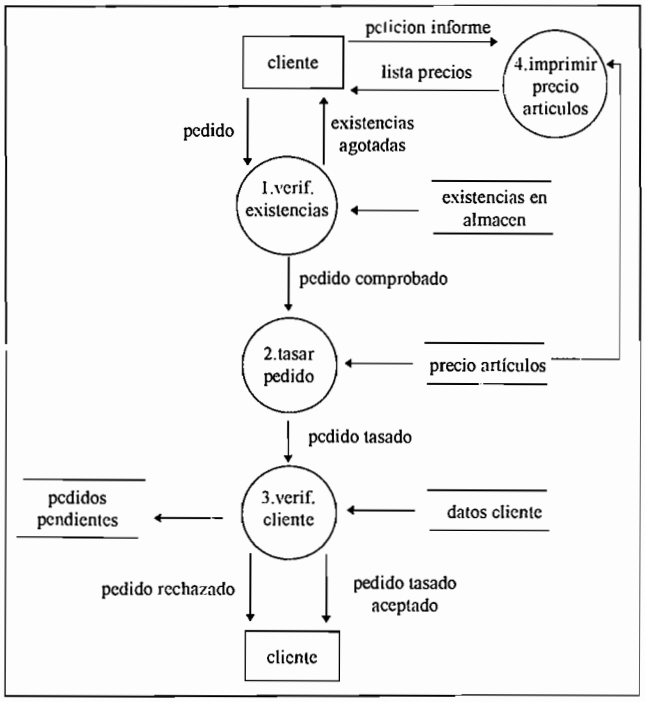


Figura 5: Un ejemplo de gestión de pedidos

A continuación, el sistema de soporte a la ejecución entra en acción (debe ejecutar el proceso 'verif. cliente', para lo cual es necesario instanciar sus flujos de datos de entrada, recuperar registros de un almacén, etc., así hasta alcanzar los flujos de datos que salen de la entidad externa 'cliente'); por último, se visualiza una lista con los resultados solicitados (variables instanciadas). Actualmente, existe una versión de este entorno disponible en PC con Arity PROLOG, que cubre la definición de requisitos, la generación automática de prototipos y la ejecución de prototipos.

3. MSL: un lenguaje de miniespecificaciones

Una especificación MSL consta de dos partes:
- definición de los componentes del sistema (entidades externas, procesos, almacenes de datos y flujos de datos);
- conjunto de miniespecificaciones de procesos.

La primera parte se genera de forma automática a partir de la especificación de los DFD previamente traducida a hechos PROLOG. En ella se incluyen todas las entradas del DD y consta de cuatro secciones:
- 'entidades externas', - 'procesos',
- 'almacenes' - 'flujos de datos'.

En las dos primeras se enumeran los identificadores de las entidades externas y los procesos según aparecen en los DFD. En las dos últimas se describen los campos componentes de los almacenes y flujos de datos según informaciones procedentes del DFD y del DD. La **figura 6** muestra parcialmente esta parte de la especificación para el DFD de la figura 5, el ejemplo de gestión de pedidos.

La segunda parte está compuesta de un conjunto de descripciones (miniespecificaciones) de las primitivas funcionales. Cada primitiva funcional tiene asociado un identificador, al que sigue una lista entre paréntesis de los flujos de datos de entrada/salida del proceso. Esta cabecera se genera de forma automática. La especificación completa de un proceso consiste en dos secciones: 'proceso' y 'generacion flujos', con la siguiente sintaxis:

```
entidades externas:
   cliente
almacenes:
   precio_articulos
     - codigo_articulo
     - articulo
     - precio_articulo
   datos_cliente
     - codigo_cliente
     - nombre_cliente
     - tipo_cliente
   pedidos_pendientes
     - nombre_cliente
     - ciudad_cliente
     - código_artículo
     - cantidad_pedida
procesos:
   verif._existencias,
   tasar_pedido,
   verif._cliente
   imprimir_precio_articulos

flujos de datos:
   pedido_comprobado
     - nombre_cliente
     - ciudad_cliente
     - articulo
     - cantidad_pedida
   pedido_tasado
     - nombre_cliente
     - ciudad_cliente
     - articulo
     - cantidad_pedida
     - precio_pedido
   pedido_tasado_aceptado
     - nombre_cliente
     - ciudad_cliente
     - articulo
     - cantidad_pedida
     - precio_final
   pedido_rechazado
     - nombre_cliente
     - ciudad_cliente
     - articulo
     - cantidad_pedida
```

Figura 6: definición MSL de algunos componentes del DFD de la figura5

PROCESO <identificador> (e_1,...,e_n s_1,...,s_m)
<descripción proceso>
GENERACION FLUJOS
<decisión>
FINPROCESO

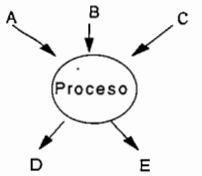
donde e_i/s_i: flujos y almacenes de datos de entrada / salida;
<descripción proceso>: sentencias estructuradas que describen las tareas del proceso.
<decisión>: precondición, opcional, asociada a cada uno de los flujos de datos generados.

Se consideraría así p.ej. flujos de datos mutuamente exclusivos. Esta característica permite, en cierto modo, usar un mecanismo de pre y pos-condiciones como recomienda [Yourdon 89], potenciando la capacidad expresiva de las ME de los procesos.

MSL utiliza una notación imperativa similar a la de los lenguajes estructurados ¿Por qué usamos una semántica imperativa en vez de declarativa (como en PROLOG o SQL)? La razón radica en nuestro interés por utilizar un vocabulario común a todos los usuarios de MÉTRICA 2 y mantenernos dentro de un estándar aunque por ello se pierda capacidad expresiva y declarativa. Las sentencias se dividen en dos grupos:

Sentencias que caracterizan a los lenguajes estructurados imperativos: ES (asignación), SI..ENTONCES..SINO..FINSI (decisión), SEGUN..SELECCIONAR..FINSEGUN (alternativa), PARA CADA..EJECUTAR..FINPARA (repetición), LEER y ESCRIBIR (entrada/salida). En la sentencia de asignación ES, el lado izquierdo debe ser un identificador o un componente de un flujo de datos. La sentencia SEGUN..SELECCIONAR se muestra útil cuando la lógica de un proceso es compleja. Las sentencias LEER y ESCRIBIR permiten a los usuarios introducir datos mediante el teclado o visualizar resultados en pantalla.

Sentencias relacionadas con características propias de los DFD, como son la gestión de almacenes y flujos de datos: RECUPERAR DE <almacén> recupera un registro del almacén; BORRAR DE <almacén> elimina un registro del almacén y ALMACENAR EN <almacén> añade un registro a un almacén; INDEFINIDO <lista flujo de datos de salida> permite dejar incompletamente definida una especificación, o sea no se dice cómo se van a generar los flujos de datos de la lista y será en tiempo de ejecución cuando se solicite al usuario que introduzca manualmente valores para los elementos de los flujos indefinidos (esta sentencia es útil cuando no es factible realizar un prototipo completo del sistema y sólo se pretende prototipar las partes de mayor interés, dejando el resto indefinidas y permitiendo al usuario aportar interactivamente un comportamiento por defecto, similar a las clases abstractas de la tecnología Orientada a Objetos). La sentencia OBTENER <lista flujo de datos entrada> (en el caso de que los flujos de datos tengan su origen en otro proceso) permite deshacer una parte importante de la ambigüedad de los DFD al hacer explícita la obtención de los valores de los flujos de datos. En el **ejemplo** se fija el control interno del proceso y se detalla en qué casos se hará uso de la información que contiene un determinado flujo de datos. En la sección GENERACION FLUJOS de un proceso es posible especificar condiciones para la generación de los flujos y determinar en qué casos se generarán los flujos de salida D y E. La utilización de la sentencia OBTENER en combinación con la sección de GENERACION FLUJOS y sentencias de control permiten definir sin ambigüedades la lógica de control interna de un proceso lo que permite la ejecución de los DFD.




```
Proceso P (A, B, C -> D, E)
...
SEGUN <identificador>
SELECCIONAR
  caso 1: GET A,
    ...
  caso 2: GET B
    ...
  caso 3: GET C, B,
FINSEGUN
.....
FIN PROCESO
```

MSL también permite el uso de operadores matemáticos básicos con el fin de escribir expresiones de cálculo pero evitando, por el número y naturaleza de los operadores que incluye, que éstas sean complejas, tal como recomienda MÉTRICA 2. La **figura 7** proporciona las especificación de dos de los procesos del ejemplo de la figura 5.

Es importante reseñar que se permite el uso de la notación punto para indicar de forma inequívoca el objeto al que se refiere una sentencia. P. ej. si un proceso tiene dos flujos de entrada que proceden de otro proceso (flujoA) y de un almacén (flujoB) y los dos flujos incluyen un campo con idéntico nombre (campoX), será preciso usar la notación punto para indicar cuando se trata del campo del flujo A (flujoA.campoX) o cuando se trata del campo del flujo B (flujoB.campoX), lo que preserva la propiedad de identificador único.

4. Generación de prototipos

Existen trabajos que muestran cómo representar los DFD y las ME en un lenguaje declarativo como PROLOG [Goble 89] [Docker 88] o PARLOG [Steer 88], aprovechando las buenas propiedades de los lenguajes lógicos para la construcción de prototipos ejecutables. En las técnicas descritas en los artículos anteriores el prototipo se construye de forma 'manual', por lo que los productores de software deben escribir directamente código en algún lenguaje lógico (PROLOG, PARLOG). Nosotros integramos un traductor de MSL a PROLOG de modo

```
PROCESO tasar_pedido (pedido_comprobado, precio_articulos → pedido_tasado) :
    OBTENER pedido_comprobado;
    RECUPERAR DE precio_articulos;      /* Un pedido = n elementos de un tipo de artículo */
    precio_pedido ES cantidad_pedida * precio_articulo;
GENERACION FLUJOS
    pedido_tasado;
FINPROCESO

PROCESO verif._cliente (pedido_tasado, datos_cliente →
                        pedido_tasado_aceptado, pedido_rechazado, pedidos_pendientes) :
    OBTENER pedido_tasado;
    RECUPERAR DE datos_cliente;
    SEGUN tipo_cliente SELECCIONAR
      MOROSO :  ESCRIBIR "pedido no aceptado";
      NORMAL :  precio_final ES precio_pedido ;
                ALMACENAR EN pedidos_pendientes;
      ESPECIAL : precio_final ES precio_pedido * 9 / 10 ;  /* Se le hace precio especial */
                ALMACENAR EN pedidos_pendientes ;
    FINSEGUN;
GENERACION FLUJOS
    SI tipo_cliente = MOROSO GENERAR
        pedido_rechazado ;
    SINO GENERAR
        pedido_tasado_aceptado ;
    FINSI;
FINPROCESO
```

Figura 7: Niniespecificaciones correspondientes a procesos de la figura 5

que los productores de software sólo necesitan aprender MSL (muy parecido a un Lenguaje Estructurado). El traductor se encarga de generar el prototipo PROLOG a partir del código MSL.

En [Goble 89] se propone una técnica manual para convertir los DFD y las ME en programas PROLOG. Nosotros utilizamos estas ideas asociando una regla PROLOG a cada flujo de datos de salida de un proceso, cuyo nombre coincide con el del identificador del flujo de datos en la especificación MSL y cuyo número de argumentos es igual al número de elementos del flujo de datos. La ejecución de la regla asignará un valor a cada uno de los elementos del flujo de datos. Suponemos que los flujos de datos que entran a un almacén o salen de él poseen idéntico nombre que el almacén. Asociamos a cada primitiva funcional ***P*** (proceso) una regla cuya cabecera es el predicado que corresponde al nombre del proceso y donde los argumentos del predicado son todos los flujos de datos de entrada y salida al proceso; el cuerpo de la regla representa las acciones del apartado <descripción del proceso> de su especificación MSL.

Un flujo de datos de salida de un proceso *P* tiene la siguiente regla asociada: su cabecera es el predicado asociado al flujo de datos con sus correspondientes argumentos (es decir, $flujoDatosSalida_i(ListArgSalida)$); su cuerpo consta de:
- el predicado correspondiente al proceso *P* ($proceso_P(ListArgProceso_P)$), que a su vez contiene en su cuerpo llamadas a los predicados correspondientes a los flujos de datos de entrada al proceso *P*; y
- los predicados correspondientes a las condiciones reflejadas en la sección GENERACION FLUJOS de *P* para el flujo de datos ($generacionFlujosProceso_P$).

$flujoDatosSalida_i$(ListArg):-
    $proceso_P$(ListArg$Proceso_P$),
    generacionFlujos$Proceso_P$.

donde $generacionFlujosProceso_P$ no aparece cuando la generación es incondicional. En el caso de una generación de flujos condicional se pueden presentar dos opciones:
- Si $flujoDatosSalida_i$ aparece en una sentencia SI..ENTONCES..SINO..FINSI, $generacionFlujosProceso_P$ equivale a la condición o a la negación de la condición, dependiendo de si el flujo se asocia con las cláusula 'ENTONCES' o 'SINO', respectivamente.
- Si $flujoDatosSalida_i$ aparece asociado a un valor *v* en una sentencia SEGUN..SELECCIONAR..FINSEGUN con etiqueta *i*, $generacionFlujosProceso_P$ equivale a la igualdad $i=v$.

Si en el cuerpo de una regla correspondiente a un proceso existe un predicado relativo a un flujo de datos generado por otro proceso, existirá otra regla en el prototipo para calcular tal flujo de datos. Otra posibilidad es que en el cuerpo de una regla se haga referencia a un flujo de datos que tiene su origen en una entidad externa o en un almacén. En el primer caso, es preciso incluir en la base de datos 'hechos' PROLOG con información acerca del flujo de datos, o bien introducir por teclado la información necesaria, en el momento de la ejecución de la regla. En el caso en el que el flujo de datos tiene su origen en un almacén, es necesario que en la base de datos haya un hecho para cada uno de los registros del almacén, de la forma: $nombre\_almacen(Valor_1, Valor_2, ...)$.

4.1. Traducción de sentencias MSL a PROLOG

La traducción de las miniespecificaciones MSL a un conjunto de cláusulas PROLOG está basada en el trabajo de [Williams 86] que trata la traducción de programas de PASCAL a PROLOG. El anexo A presenta una tabla con la correspondencia entre sentencias MSL y reglas PROLOG utilizada para generar los prototipos. El anexo B muestra un fragmento del prototipo PROLOG correspondiente a las miniespecificaciones MSL de las figuras 6 y 7.

5. Conclusiones y trabajo futuro

Hemos presentado una herramienta para generar y ejecutar prototipos funcionales a partir de especificaciones estructuradas (DFD, DD y ME) con el objetivo de solucionar la falta de herramientas para el prototipado funcional. De esta manera realizamos una aportación a la metodología oficial MÉTRICA 2 que pone énfasis en el uso de técnicas estructuradas y de prototipos funcionales para incentivar la participación de los usuarios en el proceso de desarrollo de SI, a la vez que validar/verificar sus requisitos.

Usando MSL se obtienen ME que son a la vez formales y simples: al ser formales es posible validar automáticamente la información de los DFD y del DD, generar prototipos y, además, corregir errores cometidos por los productores de software en la fase de análisis; al ser simples, permiten a los usuarios o a los programadores entender fácilmente las ME escritas por los productores de software debido, en gran parte, a las propiedades de un Lenguaje Estructurado [Vessey 86].

Nuestro interés actual se centra en la investigación de los fundamentos formales de los conceptos de AE para intentar unificar en un solo modelo las diferentes técnicas de AE a través de una teoría matemática única. Con esto se pretende mejorar la calidad de las herramientas software y de las técnicas de generación de prototipos relacionadas con AE, al permitir una validación y verificación continua de los diferentes modelos del sistema.

En relación con el lenguaje MSL, estamos trabajando en la conveniencia de distinguir entre funciones primitivas y no primitivas (jerarquías) con el propósito de permitir prototipos de alto o bajo nivel de abstracción aún cuando los procesos no estén especificados completamente: algo así como 'miniespecificaciones abstractas' en clara analogía con las clases abstractas del paradigma OO; lo que se ha revelado muy útil durante el análisis.

Bibliografía

[Docker 88] T.W.G Docker; *SAME: A Structured analysis Tool and its Implementation in Prolog.* En *Logic Programming.* Actas de la V Conference and Internat. Symposium. MIT Press, 1988.

[Ehrig et al 92] H. Ehrig; B. Mahr; I. Classen; F. Orejas; *Introduction to Algebraic Specification. Part 1; Formal Methods for Software Development.* The Computer Journal, vol. 35, nº 5, 1992, pp. 460-467.

[France et al. 89] Robert B. France y Thomas W.G. Docker; *Formal Specification using structured systems analysis.* Lecture Notes in Computer Science. ESES'89. C. Ghezzi et al. (eds.) Springer-Verlag nº387.

[Fuchs 92] Norbert E. Fuchs; *Specifications Are (Preferably) Executable.* Software Engineering Journal, septiembre 1992.

[Fuggeta et al 93] Alfonso Fuggeta, C. Ghezzi, D. Mandrioli y A. Morzenti; *Executable Specifications with Data-flow*

Diagrams. Software-Practice and Experience, Vol. 23 (6), 629-653, junio 1993.
[Goble 89] T. Goble; *Structured Systems Analysis through Prolog*. Prentice-Hall 1989.
[Goguen 94] Joseph A. Goguen; *Requirements Engineering as the Reconciliation of Technical and Social Issues*. En Requirements Engineering: Social and Technical Issues editado con Marina Jirotka, Academic, 1994, pp. 165-199.
[Harel 92] D. Harel; *Biting the Silver Bullet*. Computer, vol. 25, nº1, enero 1992, pp. 8-20.
[Larsen et al 94] Larsen P.G., Nico Plat y Hans Toeenel; *A Formal Semantics of Data Flow Diagrams*. Formal Aspects of Computing, diciembre 1994.
[Luqi 89] Luqi. Naval Postgraduate School; *Software Evolution Through Rapid Prototyping*. IEEE COMPUTER vol.22, n.5, mayo 1989 pp.13-25.
[MAP 95] *Metodología de Planificación y Desarrollo de Sistemas de Información. MÉTRICA V.2.1*. MAP: Ministerio de la Administración Pública, Sub. Gen. de Coordinación Informática. TECNOS, Madrid, 1995.
[Molina et al 92] J. García Molina, A. Toval Álvarez; M. González Rodríguez; *SAPE: A structured analysis prototyping environment*. 12Th World Computer Congress. IFIP Congress 92, Madrid 1992.
[OOAP 95] *Object-Oriented Assistant Prototyper*. Proyecto PASO (Plan de Acción SOftware) subvencionado por CEE (DG. XIII) y Mº de Industria. DIS (Univ. de Murcia) y DSIC (UPV Valencia), junio 1995.
[Rumbaugh 91] Rumbaugh, J. et al.; *Object-Oriented Modeling And Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
[Steer 88] K. Steer; *Testing Data Flow Diagrams with PARLOG*. 1988. En *Logic Programming*. Actas de la V Conference and International Symposium. MIT Press, 1988.
[Tanik et al. 89] M.M. Tanik; R.T. Yeh; *Rapid Prototyping in Software Development*. IEEE Computer vol.22, nº5, mayo 1989 (Guest Editor's Introduction).
[Tao et al. 91] Yonglei Tao y Chenho Kung; *Formal Definition and Verification of Data Flow Diagrams*. Journal of Systems Software. 1991.
[Toval et al. 94] *Prototyping Object Oriented Specifications in an Algebraic Environment*. **A. Toval, I. Ramos, O. Pastor.** En *Database and Expert Systems Applications* (D.Karagianis, Ed.). LNCS nº856. Springer-Verlag 1994.
[Tse 91] T.H. Tse; *A unifying Framework for Structured Analysis and Design Models: An Approach using Initial Algebra Semantics and category Theory*. Cambridge University Press 1991.
[Tse et al. 94] T.H. Tse, T.Y. Chen, C.S. Kwok; *The use of Prolog in the modeling and evaluation of structure charts*. Information and Software Technology 1994, Vol. 36 nº1, 23-33.
[Vessey 86] Iris Vessey and Ron Weber; *Structured Tools and Conditional Logic: An Empirical Investigation*. Comm. de ACM, junio 1986, vol.29, nº1 pp. 48-57.
[Williams 86] M.H. Williams, G.Chen; *Translating Pascal for execution on a Prolog-based System*. The Computer Journal, vol.29, nº3, 1986.
[Yourdon 89] Edward Yourdon; *Modern Structured Analysis*. YOURDON Press 1989.

SENTENCIAS DE CONTROL

Sentencia MSL	Código PROLOG

"SI..ENTONCES..SINO"	if (IN,OUT):-
	condición,
	acciones_ENTONCES,¡.
	if(IN,OUT):- acciones_SINO.

"SI..ENTONCES"	if(IN,OUT):-
	condición,
	acciones_ENTONCES,¡.
	if(IN,IN).

"SEGUN ..SELECCIONAR..FINSEGUN"	case(Etiqueta_1.IN,OUT):-acciones_1. case(Etiqueta_2,IN,OUT):-acciones_2. case(Etiqueta_n,IN,OUT):-acciones_n. case(,,):-accionesPREDETERMINADAS

"PARA CADA..EJECUTAR..FINPARA"	wile(IN,OUT):-
	condición,
	acciones_PARA,
	while(IN,OUT).
	while(IN,IN).

SENTENCIAS DE ENT/SAL Y GESTIÓN DE ALMACENES

Sentencia MSL	Código PROLOG
"LEER"	read(Identificador).
"ESCRIBIR"	write(Identificador). write(Cadenas$).
"RECUPERAR DE	"nombre_ archivo(Campo_1, Campo_2,...)
"BORRAR DE"	retract(nombre_archivo(Campo_1, Campo_2,...))
"ALMACENAR EN"	assert(nombre_archivo(Campo_1, Campo_2,...))
"OBTENER"	nombre_ flujo_ datos(Campo_1, Campo_2,...)

Anexo A. Traducción de sentencias MSL a PROLOG

```
/* pedidos de los clientes */
pedido(toval,murcia,silla,50).        pedido(gonzalez,molinaDeSegura,mesa,40).

/* existencias en almacén, precios de los artículos y datos de los clientes */
existencias_en_almacen(1,silla,150).        existencias_en_almacen(4,mesa,90).
precio_articulos(1,silla,23).               precio_articulos(4,mesa,45).
datos_cliente(2,toval,MOROSO).              datos_cliente(3,gonzalez,NORMAL).


pedido_tasado(Nombre_cliente,Ciudad_cliente,Articulo,Cantidad_pedida,Precio_pedido):-
      tasar_pedido(Nombre_cliente,Ciudad_cliente,Articulo,Cantidad_pedida,_codigo_articulo,
                   Precio_articulo,Precio_pedido).

verif_cliente(Nombre_cliente,Ciudad_cliente,Articulo,Cantidad_pedida,Precio_pedido,
          Codigo_cliente,Tipo_cliente,Precio_final):-
      pedido_tasado(Nombre_cliente0,Ciudad_cliente0,Articulo0,Cantidad_pedida0,Precio_pedido0),
      datos_cliente(Codigo_cliente0,Nombre_cliente0,Tipo_cliente0),
      case1(Tipo_cliente0, Tipo_cliente0,Precio_pedido0,Nombre_cliente0,
          Ciudad_cliente0,Articulo0,Cantidad_pedida0,Precio_final1).

case1(1,Tipo_cliente0,Precio_pedido0,Nombre_cliente0,Ciudad_cliente0,
          Articulo0,Cantidad_pedida0,Precio_final1):-
      write($No es posible aceptar el pedido$),nl.

case1(2,Tipo_cliente0,Precio_pedido0,Nombre_cliente0,Ciudad_cliente0,
          Articulo0,Cantidad_pedida0,Precio_final1):-
      Precio_final1 is Precio_pedido0,
      assert(pedidos_pendientes(Nombre_cliente0,Ciudad_cliente0, Articulo0,Cantidad_pedida0)).

case1(3,Tipo_cliente0,Precio_pedido0,Nombre_cliente0,Ciudad_cliente0,
          Articulo0,Cantidad_pedida0,Precio_final2):-
      Precio_final2 is Precio_pedido0*9/10,
      assert(pedidos_pendientes(Nombre_cliente0,Ciudad_cliente0,Articulo0,Cantidad_pedida0)).

pedido_rechazado(Nombre_cliente,Ciudad_cliente,Articulo,Cantidad_pedida):-
      verif_cliente(Nombre_cliente,Ciudad_cliente,Articulo,Cantidad_pedida,
          Precio_pedido,Codigo_cliente,Tipo_cliente,Precio_final),
      Tipo_cliente = 1.

pedido_tasado_aceptado(Nombre_cliente,Ciudad_cliente,Articulo,Cantidad_pedida,Precio_final):-
  verif_cliente(Nombre_cliente,Ciudad_cliente,Articulo,Cantidad_pedida,Precio_pedido,Codigo_cliente,
          Tipo_cliente,Precio_final),
      not((Tipo_cliente = 1)).
```

Anexo B. Fragmento de prototipo PROLOG

Ingeniería del Software

Josep Ramon Freixanet, Joan Manel Espejo, Joan Canal
Centre de Càlcul de Sabadell (CCS)

CCASE, una herramienta MetaCASE parametrizable

***Resumen**: el advenimiento de la sociedad de la información fuerza a los usuarios actuales a ampliar su soporte software en todos los campos. Es además esencial la definición de una metodología de trabajo seguida por todos los miembros de un proyecto para poder establecer correctamente su ciclo de vida. Por ambas razones han aparecido las herramientas CASE de soporte para facilitar el trabajo del ingeniero de software en el proceso de desarrollo; herramientas que deben ser adaptables a la metodología particular de desarrollo de software seguida por cualquier compañía o departamento. Este documento es una explicación breve de las características y funcionalidades principales así como del comportamiento global del prototipo final de CCASE, un proyecto financiado parcialmente por la Comisión Europea y por el Ministerio de Industria y Energía. CCASE está orientado al desarrollo de un prototipo de una herramienta MetaCASE orientada al usuario (que pueda seguir sus indicaciones) y totalmente parametrizable. El consorcio CCASE está formado por las siguientes entidades: Centre de Càlcul de Sabadell (CCS), Universitat Rovira i Virgili (URV), Barcelona Centre de CAD/CAM (BC.CAD), Institut Municipal d'Assistència Sanitària de Barcelona (IMAS) y Institut Català de Tecnologia (ICT).*

1. Introducción

En cualquier proyecto de desarrollo de software es esencial la definición de una metodología de trabajo común a seguir por toda persona involucrada en el proyecto, durante el ciclo de vida de éste. Actualmente existen muchas metodologías de desarrollo de software, conviviendo las estructuradas clásicas como MERISE, SSADM, METRICA-2, etc.; particularizaciones de estas metodologías realizadas por algunas grandes compañías de software (como la metodología desarrollada por CCS) y las que soportan las nuevas filosofías de desarrollo de software como orientación al objeto, gestión del conocimiento, sistemas cooperativos y modelo cliente-servidor.

También existe un conjunto de herramientas de soporte orientadas a ayudar al ingeniero de software en el proceso de desarrollo. Estas herramientas llamadas CASE (*Computer Aided Software Engineering*) son una de las áreas del software más prometedoras de los años 90.

Las primeras herramientas CASE aparecieron en los primeros años 80 como simples instrumentos de creación de diagramas estructurados y de ayuda a la documentación del software. A mediados de los 80 se añadieron nuevas funcionalidades, para comprobar la completitud y correctitud de los diagramas y almacenar los datos introducidos en bases de datos llamadas *Repositorios CASE*, centrándose en automatizar el diseño. A finales de los 80 se añadieron nuevas funcionalidades, como la generación automática de código a partir de las especificaciones del diseño, que establecen un enlace entre la automatización del diseño y la de la programación. Fueron emergiendo dos estándares *de facto*: AD/Cycle de IBM y PCTE a partir de los resultados del proyecto ESPRIT I32). La tendencia actual del entorno CASE se centra en la compatibilidad con las estrategias PCTE o AD/Cycle.

Las futuras herramientas CASE deberán ser adaptables a las mencionadas filosofías de desarrollo de software (orientación al objeto, gestión del conocimiento, sistemas cooperativos, modelo cliente-servidor) y ser capaces de soportar reingeniería de proyectos. Esto implica que grandes áreas diferenciadas de aplicación necesitan su metodología propia particular. Todo ello, junto al incremento de los requerimientos del usuario final para desarrollar diferentes aplicaciones siguiendo diferentes metodologías, crea la necesidad de construir una herramienta CASE totalmente parametrizable de manera avanzada e inteligente según los requerimientos del usuario final.

Siguiendo estas necesidades, el proyecto CCASE ofrece una herramienta CASE flexible para las metodologías existentes y futuras, orientada al usuario y parametrizable bajo su demanda. El proyecto CCASE está asimismo orientado a conseguir una mejora significativa del proceso de desarrollo de software europeo en todos los sectores de la economía, subiendo los niveles necesarios para realizar sistemas software de calidad y relevancia significativas, incrementando la participación europea dentro del mercado CASE y estando al día de MetaCASE, la tecnología más novedosa dentro del mundo CASE.

En la siguiente sección, se realiza una descripción de las metodologías de análisis y diseño utilizadas en el proyecto así como el marco de trabajo de éste. En la tercera sección, se ofrece una breve explicación de la arquitectura global del proyecto CCASE, centrándose en la arquitectura de los módulos CCASE existentes: MetaCASE y CoreCASE, con una sección especial para las Extended CASE Utilities existentes y futuras. Finalmente, se describen brevemente diversas conclusiones y un resumen de acciones futuras y extensiones del proyecto.

2. Herramientas de Análisis y Diseño

En esta sección se describen las metodologías y plataformas comunes de trabajo a ser utilizadas a lo largo del proyecto para optimizar el desarrollo común, el control del proyecto así como la transferencia de información a lo largo del proyecto.

Metodología del Proyecto. Se ha escogido la metodología de Booch 93 para soportar las fases de análisis y diseño del proyecto. La herramienta CASE *System Architect* ha sido utilizada como herramienta CASE para el desarrollo del proyecto, hasta disponer de versiones fiables de CCASE.

Plataforma Hardware. CCASE ha sido implementado sobre máquinas PC 486, con 16 Mb RAM, funcionando la actual versión ejecutable sobre PCs 486 con 8 Mb RAM..

Sistema Operativo. Aunque inicialmente CCASE fue desarrollada sobre *Microsoft Windows 3.1*, recientemente se ha realizado su actualización a *Windows95*, incorporando sus

mejoras a CCASE (memòria plana, *multithreading*, interface de usuario mejorada,...).

Herramientas de Desarrollo de Software. El módulo MetaCASE ha sido desarrollado utilizando el shell de sistemas expertos *Level5 Object 3.5*. Para el módulo Core CASE se ha utilizado el entorno de desarrollo *Microsoft Visual C++*. A lo largo del proyecto, la actualización de las versiones se ha realizado simultáneamente a su aparición, pasando de la versión 1.0 a la versión 2.1, pasando por la 1.5, 1.51 y 1.52.

3. Herramienta metaCASE orientada al usuario

El proyecto CCASE (*Herramienta CASE Adaptable para Tecnologías Avanzadas*) es un proyecto financiado parcialmente por el Ministerio de Industria y Energía y por la Comisión Europea dentro del marco de las acciones especiales PASO (ESPRIT 7506 PASO PC042). CCASE se inició en enero de 1994 y finalizará en su primera fase de prototipo en diciembre de 1995, realizandose un esfuerzo de aproximadamente 216.5 hombres-mes.

El Consorcio CCASE consta de cinco socios: CCS, Universitat Rovira i Virgili, BC.CAD, IMAS e ICT.

CCS (Centre de Càlcul de Sabadell) es el socio coordinador del proyecto, siendo el principal desarrollador del proyecto. *Universitat Rovira i Virgili de Tarragona-Reus (URV)* es un centro de investigación y ha contribuido en el análisis global de la aplicación, aportando su conocimiento en metodologías estructuradas clásicas y, por otra parte, ofreciendo las orientaciones principales en la definición y especificación de la gestión del conocimiento en el módulo MetaCASE.
BC.CAD (Barcelona Centre de CAD/CAM) ha contribuido al proyecto realizando tareas de diseño e implementación en el área de la interface gráfica del proyecto.
IMAS (Institut Municipal d'Assistència Sanitària de Barcelona), un instituto municipal que gestiona diversos

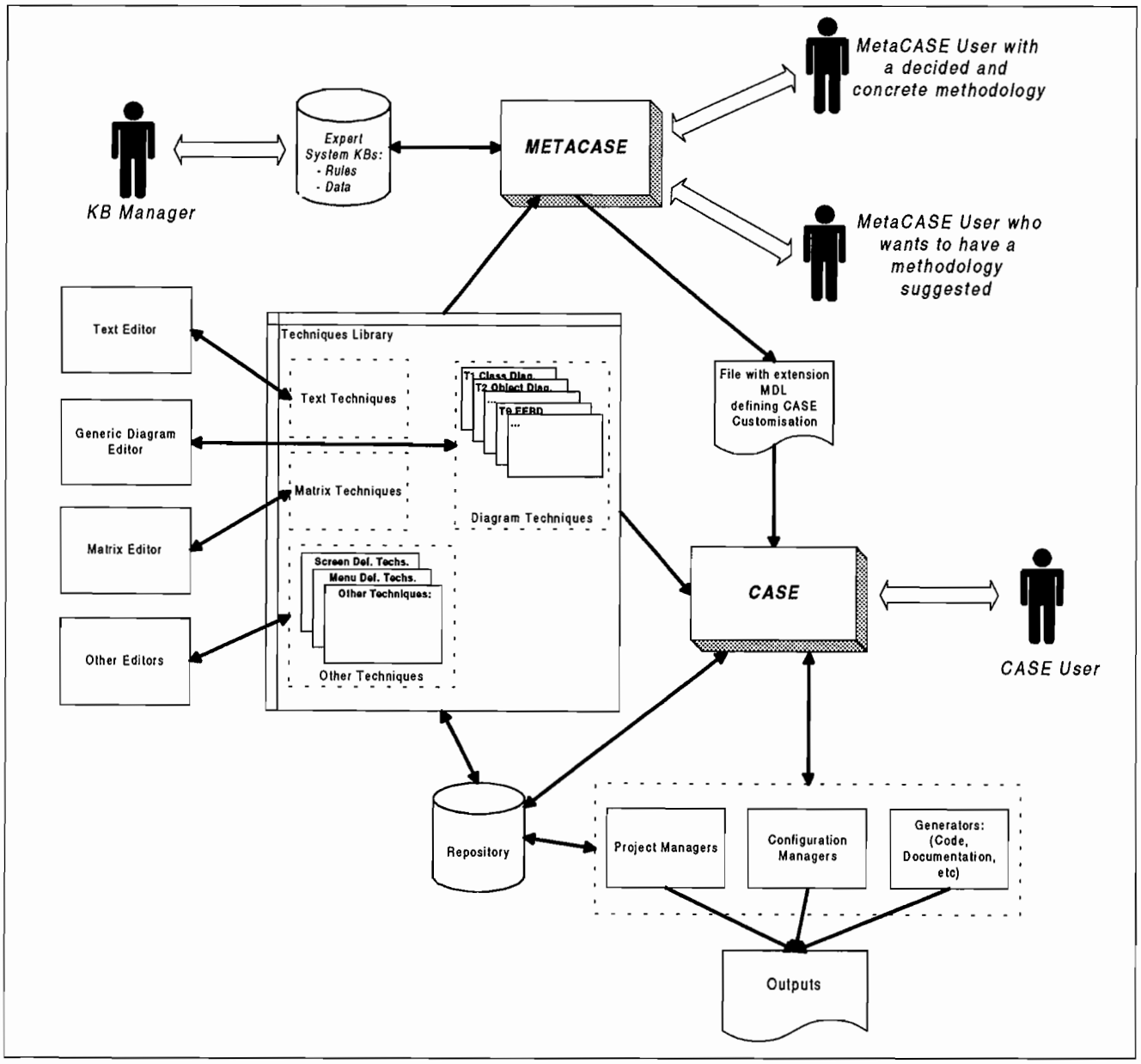


Figura 1: Arquitectura Global

hospitales en Barcelona, está desarrollando un demostrador del prototipo CCASE construyendo una aplicación Pen Computing orientada al sector hospitalario con una metodología sugerida por la misma herramienta.
ICT (Institut Català de Tecnologia) es un socio experto en el campo del control de calidad con una amplia experiencia en colaboraciones en proyectos internacionales. ICT ha contribuido al proyecto controlando la normalización y certificación del proceso de desarrollo de software en CCASE, siguiendo las recomendaciones de ISO 9000/3.

El esquema de la arquitectura global de CCASE puede verse en la siguiente**figura 1**, consistiendo en tres partes principales: un front-end parametrizador (*Intelligent MetaCASE Customizer*), un CASE Core (*MetaCASE Core*) genérico, y un conjunto de utilidades adicionales de la herramienta CASE (*MetaCASE Utilities*).

En CCASE pueden distinguirse claramente tres tipos de usuario: Usuarios CASE, Usuarios MetaCASE y Gestores de la Base de Conocimientos (KB Managers).

Los **Usuarios CASE** son los usuarios típicos de una herramienta CASE. Utilizarán su propia parametrización de CCASE dependiendo de los ficheros de parametrización previamente creados por los usuarios MetaCASE. También podrán hacer uso de todas las herramientas soportadas por CCASE, tanto herramientas de soporte al análisis y el diseño, como herramientas auxiliares (gestores de configuración o de proyecto, herramientas de testeo y varios generadores de aplicaciones).

Los **Usuarios MetaCASE** son usuarios expertos cuya principal función es la parametrización de la herramienta CASE para un entorno particular: a nivel de compañía, a nivel departamental, a nivel personal, etc., creando los ficheros de parametrización que serán utilizados a posteriori por los Usuarios CASE. Este rol será realizado por el equipo de expertos metodológicos de la compañía.

Los **Gestores de la Base de Conocimientos (*KB Managers*)** son los responsables del mantenimiento del conocimiento almacenado en el sistema experto, actualizando las bases de conocimiento y las reglas cuando sea necesario. Este rol será realizado solamente por los desarrolladores de CCASE.

CCASE ofrecerá un soporte válido a la mayoría de metodologías y entornos posibles, con un alto nivel de flexibilidad y estandarización, mediante el desarrollo de una herramienta CASE parametrizable, es decir, una herramienta MetaCASE completa. CCASE ofrece al usuario final la posibilidad de seleccionar entre un conjunto expandible de parámetros de cara a personalizar la herramienta CASE, adaptándola al tipo de aplicación a ser desarrollado. La selección de estos parámetros de construcción de la metodología puede realizarse por selección directa por parte del usuario o de manera guiada, utilizando mecanismos de inteligencia artificial para obtener la configuración o paradigma más cercano a las necesidades de la aplicación a desarrollar.

El seguimiento global del proyecto ha sido realizado en un entorno de calidad. Se ha establecido un plan de garantía de calidad y está siendo seguido desde el inicio del proyecto. Además, se está siguiendo un plan de garantía de calidad en cada desarrollo de software dentro del proyecto. Resumiendo, CCASE tiene tres partes bien diferenciadas:

El **módulo MetaCASE**, que es la parte 'inteligente' de la aplicación. Su función es ayudar al usuario a parametrizar su particular metodología, basada o no en aquellas existentes, generando un fichero de parametrización, llamado *fichero MDL*, donde se almacenará su metodología definida.

El **módulo Core CASE** genérico, que es una herramienta CASE real, que tiene un comportamiento global y su funcionamiento se particular es definido mediante la metodología creada y almacenada en el *fichero MDL*.

Junto a este módulo CASE genérico, se está realizando una adaptación de un amplio conjunto de herramienta clásicas de soporte. Estas herramientas, llamadas **MetaCASE Utilities**, son gestores de proyectos, gestores de configuración, y algunos generadores, como generadores de código, de documentación y de juegos de prueba.

Las subsecciones posteriores explican con mayor detalle estas partes de CCASE.

3.1. Intelligent MetaCASE Customizer (IMC)

El Intelligent MetaCASE Customizer es el front-end parametrizador de la herramienta. Como puede verse en la **figura 2**, consiste principalmente en un módulo inteligente formado por un motor de inferencia comercial (*Level5 Object*), con sus correspondientes bases de conocimiento conteniendo las reglas de decisión y el conocimiento global en diferentes áreas, tales como calidad, métodos, técnicas, sistemas de dibujado, plataformas, etc.,...

Como el objetivo de CCASE es crear una herramienta CASE parametrizable, la parametrización de la herramienta significa decidir que metodología debe utilizarse para desarrollar una aplicación concreta en una compañía concreta. Este proceso de decisión no es rutinario y debe ser llevado a cabo por expertos en metodologías de software. Es, pues, necesario capturar el conocimiento de los expertos e implementar su proceso de razonamiento. *Intelligent MetaCASE Customizer*

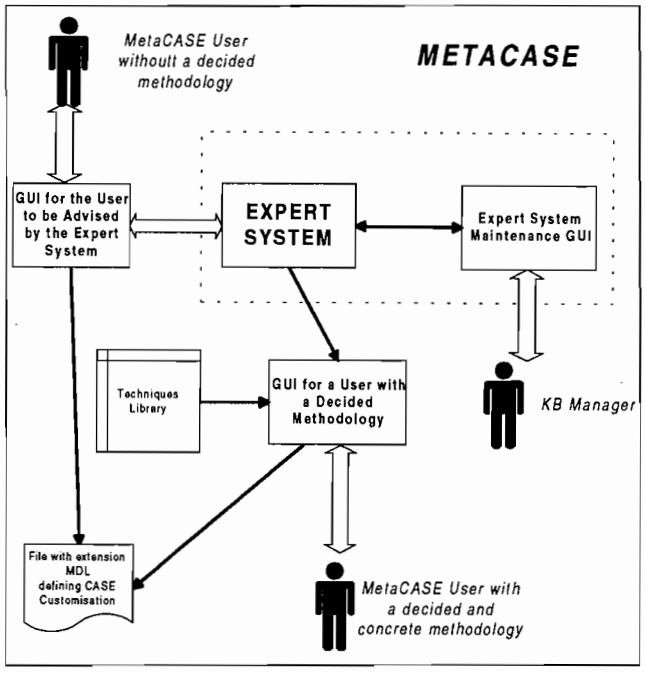


Figure 2: Arquitectura MetaCASE

(IMC) es un sistema gestor del conocimiento. Estos sistemas se llaman *Knowledge Based Systems* (KBS) y, en particular, aquellos que usan conocimiento obtenido de los expertos en tareas específicas se llaman **Sistemas Expertos**. El Conocimiento en un KBS suele representarse internamente separadamente de los mecanismos que realizan el proceso de razonamiento (Motor de Inferencia).

La función principal del IMC es proponer una metodología al usuario a partir de sus necesidades particulares. El procedimiento utilizado para recomendar una metodología está basado en entrevistar al usuario para conocer las características principales de la aplicación a desarrollar. Esta entrevista es fundamental para construir un documento de definición de requerimientos; estudiar las características de la compañía. Estas características determinan la formación de los empleados y la existencia de metodologías previas y determinar las necesidades software y hardware que la aplicación generará, teniendo en cuenta las actualmente existentes.

Este estudio muestra que en el proceso de selección de una metodología se tiene en cuenta la posibilidad de mantener metodologías previamente establecidas, aplicaciones que pueden ser reusadas, y material que puede ser reusado. Así, están definidos tres tipos de usuario en el prototipo IMC:
·Un usuario que está usando una metodología actualmente y quiere seguir usándola.
·Un usuario que quiere saber que metodología le recomienda el sistema, aunque el quiera seguir usando su metodología actual; es decir, quiere 'validar' la utilidad de la metodología que está usando.
·Un usuario que quiere ser aconsejado por IMC, usando o no una metodología actualmente.

Finalmente, es realizada por el usuario una parametrización particular de la herramienta CASE de acuerdo con sus necesidades. El proceso que el IMC ejecuta dependerá de las respuestas obtenidas del usuario. Podemos distinguir cuatro tareas en este proceso:

Recoger información del usuario. Es la fase de recogida inicial. obteniendo datos identificativos del usuario, como el nombre, la afiliación, la categoría, y otros datos básicos. Estos datos serán útiles para mantener un histórico de la consulta.
Decidir la conveniencia de un cambio de metodología. Debe decidirse si es conveniente o no el cambio de la metodología existente en el entorno de trabajo del usuario o, por el contrario y dadas las características de entorno del desarrollo, es mejor mantenerla.
Aconsejar una metodología. Es la tarea clave de todo el sistema, en la cual el conocimiento obtenido de los expertos juega un rol decisivo.
Adquirir la metodología del usuario. En el caso de que la metodología presente se mantenga, es conveniente capturarla para poder parametrizar la herramienta CASE con la metodología que el usuario desea para trabajar.

El conocimiento necesario para realizar cada una de estas tareas es definido mediante reglas. Primeramente, el sistema establece un diálogo con el usuario para obtener de él el conocimiento necesario. Cuando el sistema no conoce el valor de algunos atributos necesarios para verificar la premisa de una regla, los adquirirá preguntando al usuario. La manera de adquirir la información necesaria del usuario variará dependiendo de la tarea a realizar. Por ejemplo, en la fase de aconsejar una metodología, el sistema permitirá al usuario entrar un valor porcentual definiendo la relevancia que un conjunto de características tiene en las aplicaciones a ser desarrolladas con la herramienta CASE final. Las figuras siguientes ofrecen una idea del aspecto de este diálogo.

En la fase de adquisición de una metodología, el sistema experto establece una etapa de diálogo con el usuario. En este caso, el tipo de respuestas esperado consta de opciones tales como: *siempre, casi siempre, frecuentemente, a veces, casi nunca, nunca*. La **fig. 4** da una idea del aspecto de las pantallas permitiendo la interacción con el usuario en esta fase.

Cuando el sistema dispone de la información necesaria, instancía la metodología resultante y, finalmente, la presenta al usuario de la forma que puede verse en la **figura 5**. El usuario dispone de total flexibilidad de modificar la metodología obtenida añadiendo nuevas técnicas o eliminando algunas de las propuestas por el sistema.

En el momento en que las técnicas y elementos han sido escogidos, es posible consultar un posible ciclo de vida. Puede obtenerse un ciclo de vida sugerido a partir de las técnicas seleccionadas. Este ciclo de vida puede ser modificado por el Usuario MetaCASE, añadiendo nuevas fases o eliminando algunas de las sugeridas, para obtener un ciclo de vida personalizado. La **figura 6** muestra un ejemplo de este editor del ciclo de vida.

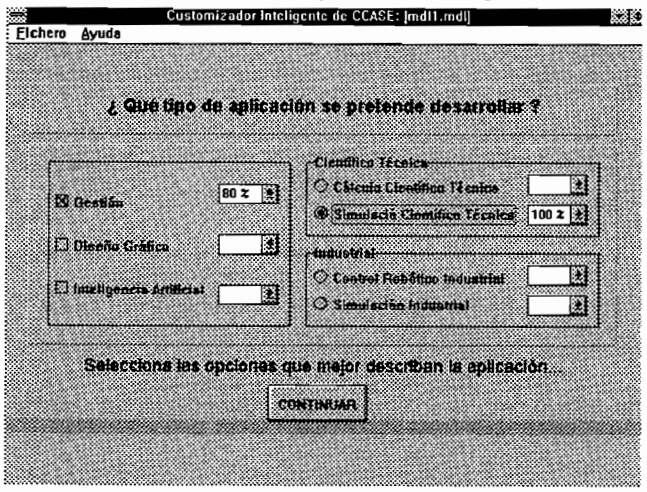


Fig. 3: Vista de la interface de usuario en la fase de aconsejar una metodología

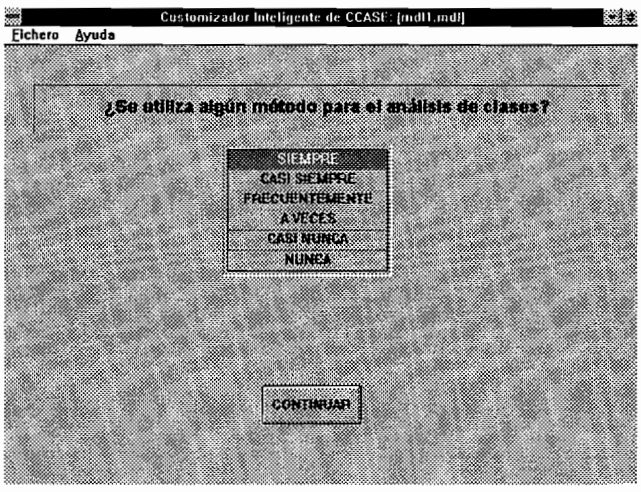


Fig. 4: Vista de la interface de usuario en la fase de adquisición de la metodología

Se ha añadido una utilidad para poder ver gráficament el ciclo de vida propuesto de cara a ayudar al Usuario MetaCASE en la parametrización del ciclo de vida. La **figura 7** muestra un ejemplo de esta utilidad gráfica.

Una vez finalizada la definición del paradigma por parte del Usuario MetaCASE, utilizando el sistema experto creado, se genera un fichero de parametrización, y este fichero '*.MDL' es utilizado para configurar el MetaCASE Core.

3.2. MetaCASE Core

MetaCASE Core es una serie de clases que colaboran en la implementación de la herramienta CASE genérica. MetaCASE Core debe verse como un **motor genérico** alimentado mediante una metodología definida en un fichero de parametrización previamente generado por el módulo *Intelligent MetaCASE Customizer*. Tiene una interacción muy estrecha con las MetaCASE Utilities para permitir un posible uso de la gestión de la configuración, gestión de proyectos y generación de código durante los procesos de análisis y diseño.

Este **motor genérico** es capaz de tratar con toda la información generada en el análisis y el diseño de una aplicación y ha sido realizado utilizando clases C++. Así mismo, ha sido implementada una completa y genérica Interface Gráfica de Usuario para el acceso y modificación del análisis y el diseño de la aplicación a desarrollar utilizando CCASE.

Las **figuras 8,9,10,11 y 12** muestran algunas imágenes del actual prototipo del MetaCASE Core.

3.2.1. Conceptos Globales

Los tres principales conceptos que se encuentra en el MetaCASE Core son: Proyectos, Técnicas y Elementos.

Los **Proyectos** contienen toda la información relativa a la aplicación a ser desarrollada. Entre esta información pueden encontrarse todos los diagramas que describen el proyecto y una marca de tiempo. Una definición del perfil del usuario se describe también para poder implementar un control de acceso. Se permite que diferentes usuarios lean o modifiquen diferentes partes del proyecto.

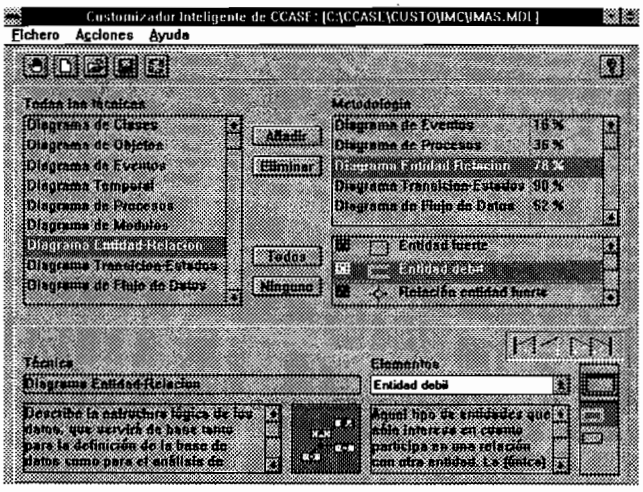


Fig. 5: Aspecto de la pantalla final de presentación de la metodología en el IMC

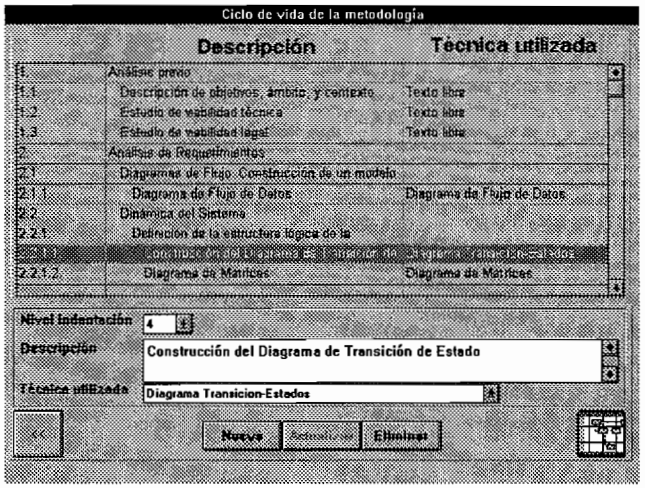


Figura 6: Ciclo de Vida Propuesto

Las **Técnicas** (diagramas, tablas y textos) son una parte importante dentro de los proyectos. Describirán el análisis y el diseño de la aplicación. Están representados los diferentes tipos de técnicas, tanto las pertenecientes a metodologías estructuradas clásicas como a metodologías de Orientación al Objeto. Por ejemplo, entre las técnicas estructuradas se encuentran los Diagramas Entidad-Relación Extendidos, Diagramas de Flujo de Datos, Diagramas de Flujo de Control y la Normalización, entre otras; entre las Orientadas al Objeto se representan Diagramas de Clases y Diagramas de Objetos. entre otros; y, finalmente, existen diagramas pertenecientes a los dos tipos de metodologías, como los Diagramas de Transición de Estados. CCASE ofrece al Usuario CASE un amplio conjunto de técnicas para describir los diferentes aspectos de una aplicación, tanto desde el punto de vista lógico como físico, y desde el punto de vista estático como dinámico de la aplicación.
Los **Elementos** son los componentes de las técnicas. Mediante adecuada parametrización, distintas metodologías pueden especificar diferentes elementos de una técnica entre el conjunto correcto de elementos de la técnica. Esta característica permite al Usuario CASE usar un subconjunto de todos los elementos de una técnica, evitando excesos de notación.

3.2.2. Arquitectura

Un resumen del esquema de la arquitectura global del MetaCASE Core y su integración con las MetaCASE Utilities

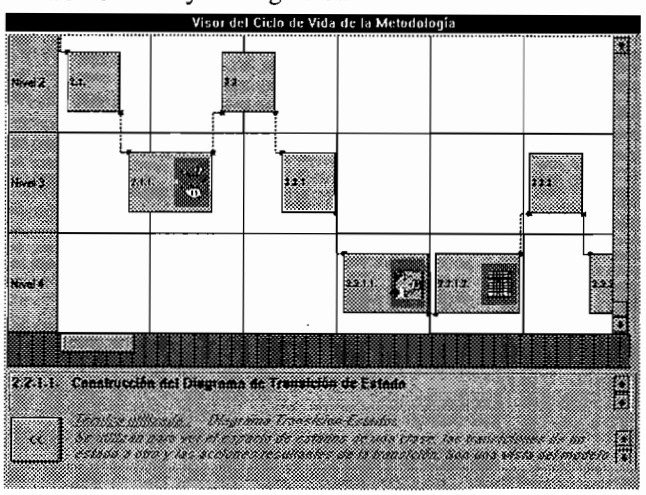


Figura 7: Representación Gráfica del Ciclo de Vida Propuesto

puede verse en la siguiente **figura 12**. El módulo MetaCASE Core puede dividirse en cuatro partes principales: **Clases Genéricas** (incluyendo Clases Gráficas Genéricas), **Gestores** (Managers), **Conexión con el *Intelligent MetaCASE Customizer***, **Librería de Técnicas** (incluyendo el Soporte Gráfico a las Técnicas). A continuación se incluye una breve explicación técnica de cada una de las partes.

3.2.2.1. Clases Genéricas (incluye Clases Gráficas Genéricas)

Son las clases que configuran el motor genérico del MetaCASE Core, es decir, la herramienta CASE genérica. Son esencialmente clases que representan el comportamiento común de los diferentes tipos de técnicas y elementos, clases representando el comportamiento del proyecto y la metodología, y un conjunto de clases auxiliares que completan las responsabilidades. Deben incluirse aquí las clases gráficas genéricas que realizan la interconexión entre estas clases genéricas y el entorno gráfico de la aplicación, incluyendo los diferentes editores genéricos (editor genérico de diagramas, de textos,...).

3.2.2.2. Gestores (Managers)

Son un conjunto de clases cuyo propósito principal es realizar el control del comportamiento global de todas las clases involucradas en el proyecto. Actúan de gestores de la aplicación. Los más importantes son:

Internal Project Manager: Proporciona una interface de control del uso de las clases genéricas y de la librería de técnicas por parte de agentes externos.
Repository Manager: Controla la aplicación de la persistencia a las técnicas y elementos involucrados en un proyecto diseñado utilizando CCASE, gestionando el Diccionario de Datos asociado a CCASE mediante ODBC.
Utilities Manager: Gestiona el control de la conexión de CCASE con las MetaCASE Utilities internas y externas, tal como se explicará posteriormente.

3.2.2.3. Conexión con el Intelligent MetaCASE Customizer

La conexión existente entre el módulo *Intelligent MetaCASE Customizer* y el módulo MetaCASE Core se realiza a través de los ficheros de parametrización ("*.MDL"). Se ha definido un metalenguaje capaz de expresar la parametrización de una metodología y, por su parte, el MetaCASE Core tiene un módulo parser capaz de entender ficheros escritos siguiendo este metalenguaje y, a partir de ahí, particularizar el MetaCASE Core para el caso concreto previamente establecido por el Usuario MetaCASE.

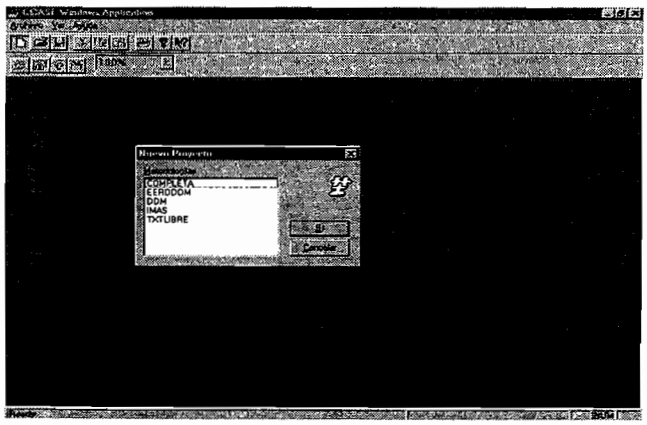


Figura 8: Carga de la Metodología de Trabajo

3.2.2.4 Librería de Técnicas (incluyendo el Soporte Gráfico a las Técnicas)

La librería de técnicas es una colección de clases que implementan las diferentes técnicas que los analistas pueden utilizar con la herramienta CoreCASE.

Estas clases se dividen básicamente en dos grupos principales: Un primer grupo consistente en las clases que conforman la implementación de las técnicas reales utilizadas por el usuario de la herramienta CASE, que se componen básicamente de los componentes de las diferentes metodologías obtenidas mediante la herramienta MetaCASE.

El otro conjunto de clases implementa el conjuntp de técnicas genéricas, las cuales representan conceptos que son comunes a las diferentes técnicas y ofrecen sus funcionalidades comunes. Existen además el grupo de clases gráficas que ofrecen la representación gráfica de todos los elementos (componentes que son combinados en las técnicas). La genericidad en el diseño de CCASE ha sido un objetivo fundamental, para fácilitar una mayor expansibilidad.

3.3. MetaCASE Utilities

El módulo *MetaCASE Utilities* está diseñado como un módulo capaz de proveer a CCASE del conjunto de utilidades existentes en la mayoría de las herramientas CASE del mercado, mediante la adaptación de utilidades existentes a CCASE o, si no es posible su adaptación, su construcción.

La interacción entre estas utilidades externas y el *MetaCASE Core* es muy fuerte. De hecho, pueden considerarse propiamente extensiones de éste, aportando una mayor funcionalidad al sistema.
Permiten ofrecer al usuario un conjunto de herramientas de ayuda en el desarrollo de un proyecto tales como: **Gestores de Configuración** (mediante la adaptación a herramientas del tipo Ms-SourceSafe, PVCS, etc.); **Gestores de Planificación de Proyectos** (permitiendo la generación de una planificación del desarrollo orientativa a partir de los resultados del IMC); **Generadores de Documentación** (sobre Ms-Word), **de Código** (Esquemas de Bases de Datos y código C++), **de Juegos de Prueba** (no soportado en la presente versión), ...

El control por parte del *MetaCASE Core* sobre estas utilidades se realiza a través del *Utilities Manager* y siguiendo en lo posible la filosofía OLE Cliente/Servidor.

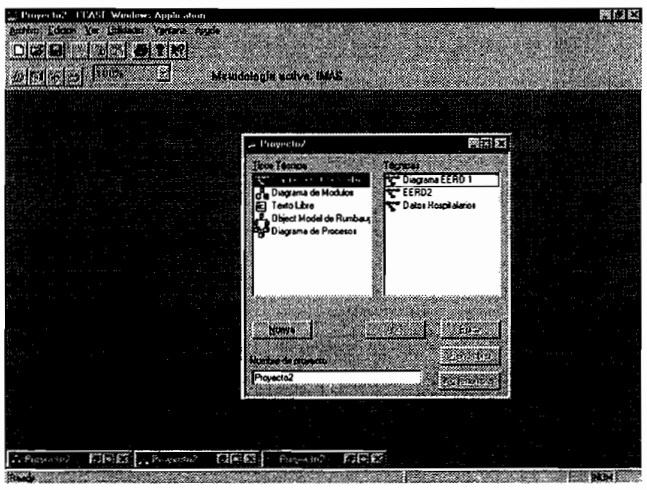


Figura 9: Escoger una Técnica a partir de una Metodología

4. Conclusión y acciones futuras

CCASE es un proyecto cuyo objetivo es la construcción de un prototipo de herramienta CASE totalmente parametrizable a partir de las orientaciones del usuario final. Es una herramienta MetaCASE completa que consta de tres partes principales: el *Intelligent MetaCASE Customizer*, el *MetaCASE Core* y las *MetaCASE Utilities*. El Intelligent MetaCASE Customizer es un front-end experto para la ayuda al usuario a definir su propia metodología, dependiendo de sus necesidades particulares. MetaCASE Core es un motor de herramienta CASE genérico, con toda la funcionalidad de una herramienta CASE, pero totalmente parametrizable mediante un fichero. Finalmente, las MetaCASE Utilities son un conjunto completo y ampliable de utilidades CASE usuales, tales como gestores de proyecto y de configuración, y diferentes herramientas de generación (código, documentos, etc.). Resumiendo, CCASE es un proyecto activo que finalizará a finales de Diciembre de 1995. Está prevista la realización de un workshop al finalizar del proyecto para su presentación en el mercado europeo del software. Está así mismo planificada una futura comercialización de la herramienta final, después de un periodo de evolución del prototipo a una versión comercial.

Referencias

[ALBER94] Albero, J.L. et al; *Análisis Orientado a Objetos del Módulo Core de CASE y la Librería de Técnicas*, PASO PC042 CCASE proj.deliv., Doc.Num: WP2-1/30-Junio-1994/02, 1994.

[BOOCH93] Booch, G.; *Object Oriented Analysis and Design with applications*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, USA, 1993.

[BRUMB94] Brumbaugh, D.E.; Object-Oriented Development. Building CASE Tools with C++, Wiley Professional Computing, 1994.

[BOTEL88] Botella, P.; *Una aproximación al CASE*, Novática, pp. 43-52, Vol.XIV, N.76, 1988.

[CHEN92] Chen, M., Norman, R.J.; *A Framework for Integrated CASE*, IEEE Software, pp. 18-22, March 1992.

[COADY91] Coad P., Yourdon E.; *Object-Oriented Analysis*, Yourdon Press, Prentice Hall, 1991.

[COADY91] Coad P., Yourdon E.; *Object-Oriented Design*, Yourdon Press, Prentice Hall, 1991.

[FRAU94] Frau, J.A., Espejo, J.M., Freixanet, J.R.; *Anexo Técnico CCASE. Herramienta CASE Adaptable para Tecnologías Avanzadas*, PASO PC042 CCASE proj.deliv, Doc. 940412-2, 1994.

[GUTER92] Guterl, F.V.; *Estándares europeos para CASE*, DATAMATION, pp. 16-17, Jul-Aug 1992.

[JARKE92] Jarke, M.; *Strategies for Integrating CASE Environments*, IEEE Software, pp. 54-61, March 1992.

[KEMER92] Kemerer, C.F.; *How the Learning Curve Affects CASE Tool Adoption*, IEEE Software, pp. 23-28, May 1992.

[LOPEZ94] López, B., Ordoyo, I.; *Especificación de los Requisitos (Análisis) de Intelligent Metacase Customizer.*, PASO PC042 CCASE proj.deliver., Doc.Number: WP1-1/30-Junio-1994/01, June 1994.

[MACG92] McGregor, J.D., Sykes, D.A.; *Object-Oriented Software Development: Engineering Software for Reuse*, Van Nostram Reinhold, 1992.

[OVUM92] *Ovum Software Europe*, Ovum Ltd., London, 1992.

[PARSA88 Parsaye, K., Chignell, M.; *Expert System for Experts*, Ed. John Wiley & Sons, Inc., New York, USA, 1988.

[PROPO93] CCASE: *Herramienta CASE Adaptable para Tecnologías Avanzadas*, PASO Project Proposal, June 1993.

[RUMB91] Rumbaugh, J. et altres; *Object-Oriented Modeling and Design*, Prentice-Hall International, Inc., 1991.

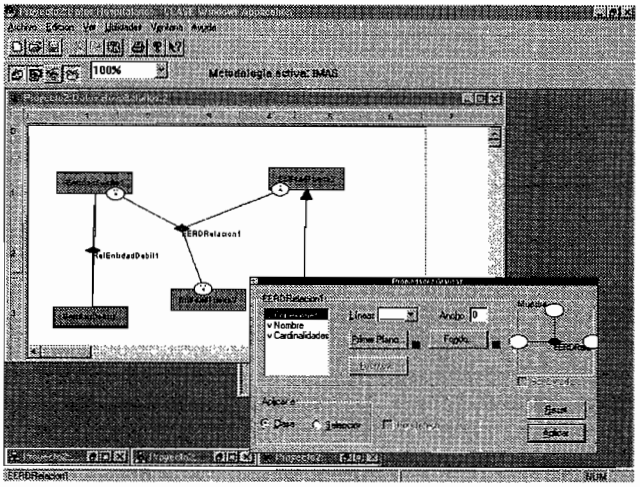


Figura 11: Edición de un Diagrama EERD

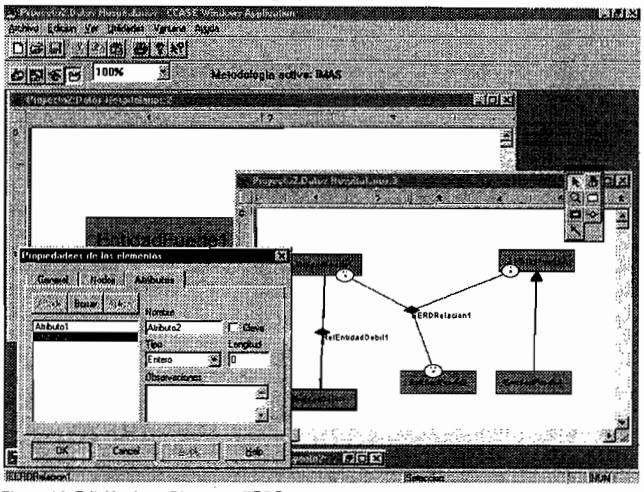


Figura 10: Edición de un Diagrama EERD

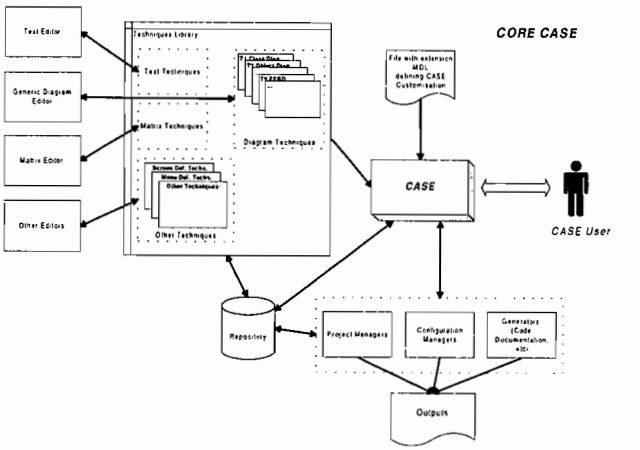


Figura 12: Arquitectura MetaCASE Core

Ingeniería del Software

Instrumentación de Modelos Reactivos para la Evaluación del Comportamiento

Alberto Valderruten Vidal, Manuel Vilares Ferro, Jorge Graña Gil
Laboratorio de Fundamentos de la Compurtación e Inteligencia Artificial , Dpto. de Computación
Universidad de la Coruña
Campus de Elviña, 15071 La Coruña, España
{valderruten,vilares,grana@dc.fi.udc.es

Resumen: *La modelización con sistemas reactivos permite describir la descomposición modular de sistemas cuyo funcionamiento implique interacciones deterministas, como los que se presentan en general en los sistemas tiempo real y en algunas entidades de comunicación. Sin embargo, para poder realizar estudios predictivos del comportamiento del sistema, es necesario ampliar la cobertura de este análisis y tomar en cuenta situaciones de no determinismo. En este artículo planteamos una estrategia para incluir una técnica de modelización del comportamiento en la metodología de diseño con sistemas reactivos propuesta por la herramienta* ***AGEL*** *en base a la utilización del lenguaje* ***ESTEREL****. Esta propuesta consiste básicamente en la inclusión de constructores cuantitativos para la definición del tiempo y de las probabilidades, completando con ellos la descripción de los aspectos puramente funcionales del sistema. Un mecanismo de monitorización permite entonces derivar resultados cuantitativos sobre el comportamiento del sistema (tiempos de respuesta principalmente), que se obtienen mediante la simulación implementada por* ***AGEL****. Aplicamos esta metodología al estudio de un protocolo de comunicación sencillo, comparando las medidas de su comportamiento con los resultados de referencia conocidos.*

Palabras Clave: *Ingeniería del Comportamiento, Modelos Reactivos Síncronos, Sistemas Tiempo Real y Acoplados, Modelización, Instrumentación de modelos, Simulación, Monitorización.*

1. Introducción

El desarrollo de sistemas con la ayuda de métodos de modelización formales viene conociendo un constante crecimiento, motivado principalmente por la posibilidad de definir el sistema de una manera precisa, sin ambigüedades, lo que permite completar el proceso de desarrollo con una etapa de validación y verificación.

Los modelos reactivos síncronos se usan para describir sistemas que deben reaccionar *instantáneamente* a estímulos emitidos por su entorno, modificando su estado interno y produciendo un evento de salida. El secuenciamiento de entradas y salidas determina el comportamiento del programa, en vez de lo contrario: un evento no puede ser pospuesto porque no sea esperado. Ejemplos de sistemas reactivos síncronos son el control de las instrucciones operativas que se emiten hacia el sistema acoplado a un satélite o el controlador de las alarmas producidas por un conjunto de sensores en sistemas críticos.

Sin embargo, este método de modelización no permite soportar análisis cuantitativos, necesarios entre otras cosas para poder realizar estudios predictivos del comportamiento del sistema. Las técnicas de evaluación del comportamiento ofrecen información sobre el comportamiento estimado de cada alternativa de diseño según criterios no sólamente funcionales, lo que constituye una base rigurosa para las decisiones que se deban tomar a lo largo del ciclo de vida del sistema. El comportamiento está definido por el IEEE (6) como el grado según el cual un sistema cumple con sus funciones en relación a unas exigencias definidas sobre cuantificadores como la velocidad, la fiabilidad o el uso de los recursos.

La complejidad creciente de los sistemas cooperativos en general, incluyendo la de los sistemas tiempo real y acoplados, motiva la integración de técnicas de evaluación del comportamiento y de descripción formal. De esta manera, se pretende abordar la especificación, implementación y análisis de dichos sistemas tomando en cuenta criterios funcionales y no funcionales (12). Al incluir información cuantitativa en los modelos formales estamos permitiendo la obtención de modelos del comportamiento capaces de estimar la calidad del servicio (velocidad, fiabilidad, ...) desde las primeras etapas del ciclo de desarrollo del sistema. Para poder llevar a cabo esta estrategia, debemos definir y validar reglas que permitan una derivación inmediata de los modelos de comportamiento.

Con este trabajo nos proponemos deducir resultados sobre el comportamiento del sistema a partir de modelos reactivos síncronos. Estos modelos describen procesos deterministas y fuertemente acoplados, siendo la comunicación realizada por un mecanismo de difusión instantánea, pero ignoran los aspectos temporales y probabilí}sticos del sistema, y por lo tanto no dan información con respecto al comportamiento. Con el fin de conseguir este objetivo, la idea es considerar las especificaciones de comportamiento de un diseño mediante *constructores cuantitativos* que permitan verificar las exigencias de comportamiento por simulación. La semántica de estos constructores, al apartarse de la de los sistemas reactivos síncronos, debe estar claramente defininida con el fin de evitar cualquier ambigüedad.

Hemos elegido una serie de trabajos conocidos con el fin de establecer una referencia para la validación de nuestros modelos. Todos están definidos sobre el uso de técnicas de descripción formal como Lotos (10,13) o las redes de Petri (9,7). Derivando un modelo de comportamiento a partir de un modelo reactivo síncrono, y resolviéndolo por simulación, debemos obtener los mismos resultados que los calculados analíticamente en los trabajos de referencia. Nuestra metodología se implementa con la ayuda del lenguaje **ESTEREL** (5) para la realización de modelos reactivos síncronos, y de la herramienta **AGEL** (1) que soporta toda la metodología de modelización. En (14) presentamos los primeros resultados. Aplicaremos esta metodología al estudio de un protocolo de comunicación sencillo, al que llamaremos *Stop & Wait*.

2. Modelos Reactivos Síncronos con ESTEREL

El lenguaje **ESTEREL** fue diseñado para implementar *modelos reactivos síncronos* (3). En estos modelos, un módulo reacciona *instantáneamente* a cada evento de entrada modificando su estado interno y creando eventos de salida. Las reacciones son síncronas con las entradas: el procesamiento de un evento es ininterrumpible, y el sistema

es suficientemente rápido como para procesar los eventos de entrada. Un sistema así posee una máquina de estados finitos implícita. Buenos ejemplos de sistemas reactivos son los controladores tiempo real *acoplados* o *embarcados* (esto es, *a bordo* del dispositivo controlado) y las entidades que componen un protocolo de comunicaciones.

ESTEREL es un lenguaje concurrente imperativo, con control de alto nivel y constructores de manipulación de eventos. Los módulos son las entidades básicas que pueden interactuar o bien con el entorno, o bien con otros módulos. Cada módulo posee una interfaz de eventos utilizada para esta comunicación. El particular estilo de modularidad en Esterel permite que los módulos puedan identificarse con los dispositivos físicos, de tal manera que su interfaz hace referencia a las señales que entran y salen de dicho dispositivo. Los eventos son los principales objetos manipulados por los módulos: pueden componerse de varias señales, que a su vez pueden estar presentes o ausentes durante la reacción a una entrada. El constructor **present** permite determinar la presencia de una señal en el evento actual:

present <señal> [then <*instrucción 1*>]
[else <*instrucción 2*>]
end

Para estar presente en una reacción, un módulo puede emitir una señal de salida gracias a la instrucción **emit** < *señal* >. Esta emisión es *difundida* a todos los módulos que tengan definida la señal como entrada. De esta manera las señales son utilizadas para especificar situaciones de sincronización y de comunicación. Las instrucciones ESTEREL son de dos tipos:

- *Instrucciones instantáneas*, basadas en la hipótesis de la sincronía perfecta, que son completamente ejecutadas en la misma reacción en la que son activadas. La emisión de señales (emit), el test de presencia de señales (present) y la secuencia (<*instrucción 1*>; <*instrucción 2*>) son ejemplos de tales instrucciones.
- *Instrucciones de espera*, la más sencilla de las cuales es el constructor **await** <*señal*>. Su ejecución se bloquea hasta la próxima ocurrencia del evento especificado.

El operador paralelo || permite la transmisión simultánea del flujo de control a las instrucciones de sus operandos. Termina de ejecutarse cuando sus dos ramas se terminan, y una vez más, no consume tiempo por sí mismo.

La instrucción **watching** permite expresar la interrupción de un comportamiento *(preemption)*, una de las funcionalidades básicas de ESTEREL. El constructor completo se define mediante la sintaxis:

do <*instrucción 1*>
watching <*señal*>
timeout <*instrucción 2*>
end

nothing
halt
emit <*señal*>
<*instrucción 1*>; <*instrucción 2*>
loop <*instrucción*> end
present <*señal*> then <*instrucción 1*> else <*instrucción 2*> end
do <*instrucción*> watching <*señal*>
<*instrucción 1*> || <*instrucción 2*>
trap <*excepción*> in <*instrucción*> end
exit <*excepción*>
signal <*señal*> in <*instrucción*> end} \\

Figura 1: Núcleo de ESTEREL

dónde la instrucción 1 será interrumpida y la instrucción 2 ejecutada si una ocurrencia de la señal tiene lugar antes de que termine la instrucción 1. Este comportamiento se termina con la instrucción 1 si no llega el *timeout*, o con la instrucción 2 si la interrupción tiene lugar. En general, las instrucciones más próximas al usuario y por lo tanto con una abstracción de más alto nivel, son derivadas de un conjunto de primitivas que constituyen el *núcleo* de ESTEREL(2) (**figura 1**). Finalmente, precisemos que aunque el compilador ESTEREL tiene previsto trabajar como *front-end* de diferentes lenguajes, utilizamos inicamente el interfaz con el lenguaje **C** para instrumentar los modelos reactivos síncronos con el objetivo de adecuarlos a la ingeniería del comportamiento.

3. Especificación del protocolo *Stop & Wait*

Consideremos el diseño del nivel de datos del modelo de referencia OSI (8). El problema está en encontrar un algoritmo para asegurar una comunicación eficiente entre dos nodos conectados físicamente mediante un canal de comunicación. Los datos se transmiten en una sola dirección, del nodo A al nodo B. El emisor envía la información en paquetes, mientras que el receptor envía únicamente un acuse de recibo por cada paquete que recibe correctamente. El emisor solo enviará un nuevo paquete una vez reciba el acuse correspondiente al paquete anterior. Puesto que el canal de comunicación puede alterar un paquete, cuando esto ocurra el paquete erróneo debe eliminarse; tras una espera adecuada sin recibir el acuse correspondiente, el emisor vuelve a enviar el mismo paquete; el proceso acaba cuando éste llega correctamente a su destino.

Los acuses de recibo que envía el receptor pueden igualmente perderse sin que el emisor pueda tener conocimiento sobre este hecho. Una vez el tiempo de espera (*timeout*) transcurrido, el mismo paquete será retransmitido, por lo que el receptor puede recibir paquetes correctos pero repetidos. Para permitir que el protocolo filtre los paquetes antes de transmitirlos al nivel superior, el receptor debe poder distinguir entre una retransmisión y un paquete correcto que llega por primera vez. La solución más sencilla consiste en manejar un número de secuencia que el emisor incluirá en el encabezado de los paquetes y que el receptor consultará para determinar su orden de llegada. Un solo bit es suficiente para determinar el número de secuencia de los paquetes, por lo que a este tipo de protocolo se le conoce con el nombre de *bit alterno* (11).

Usando una estrategia de especificación orientada recursos, modelizamos el protocolo mediante tres procesos: un emisor

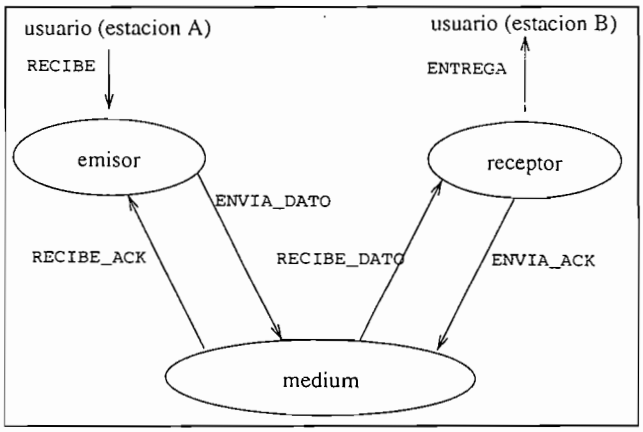


Figura 2: Modelo de Comunicaciones

(EMISOR), un receptor (RECEPTOR) y un canal de comunicación (MEDIUM). Con ESTEREL los componentes se describen en paralelo (**figura 3**). Los módulos reactivos EMISOR y RECEPTOR se describen en la **figura 4**.

En este estudio queremos poner de relieve una característica del canal de comunicación: la posible pérdida de paquetes en los dos sentidos. Asimilaremos los paquetes erróneos con los paquetes perdidos. Asimismo, no entraremos a describir en detalle las operaciones de control de paquetes duplicados, numeración y verificación de paquetes... El resultado de esta abstracción se refleja en el módulo de la **figura 5.**

En este modelo, los eventos relacionados con el tiempo (TIMEOUT, TIEMPO_DE_TRANSMISION) y la probabilidad de pérdida (ERROR) son declarados como entradas externas, lo que corresponde a un punto de vista puramente funcional, que no incluye el comportamiento del sistema en el mismo modelo. Lo que buscamos es una manera de añadir al modelo esta información sobre el comportamiento cuantitativo del sistema para permitir, gracias a las posibilidades de simulación de Agel, el cálculo de las medidas de comportamiento útiles.

4. Evaluación del Comportamiento del Protocolo

La consideración del comportamiento del sistema durante su ciclo de vida se realiza mediante un proceso de *modelización del comportamiento* que incluye un conjunto de actividades tanto de diseño del sistema como de diseño de modelos (4). En este caso, la motivación de la modelización lo constituye la existencia de *exigencias de comportamiento* que deben definirse sobre medidas concretas como tiempos de respuesta, velocidad o tasas de utilización.

Nuestra propuesta consiste en completar en una primera etapa la especificación funcional con toda la información cuantitativa (no funcional) sobre el comportamiento. Para poder comparar los resultados que producirá la simulación para este modelo con los que se obtienen a partir de redes de colas derivadas de LOTOS (13), redes de Petri temporizadas (9) o estocásticas (7) y LOTOS estocástico (10), tomaremos en cuenta los mismos valores iniciales de estos estudios. Así, el resultado que buscamos para el modelo del protocolo *Stop & Wait* es su *throughput* con las hipótesis siguientes:
- Línea de 9600 bps.
- Los paquetes de información y los acuses de recibo son de 1024 bits; el tiempo de transmisión de cada paquete es entonces de 106,7 ms.
- El próximo paquete de información está disponible tan pronto como el paquete previo ha sido enviado: solo se incluirá un lapso mínimo de 1 ms.
- El canal puede perder paquetes de información o acuses de recibo con una probabilidad del 5\%.

```
module PROTOCOLO
inputTIMEOUT,ERROR_CANAL,TIEMPODETRANSMISION;
output RECIBE, ENTREGA;
signalENVIA_DAT,ENVIA_ACK,RECIBE_DATO, RECIBE_ACK
in
            run EMISOR
      ||
            run EMISOR
      ||
            run RECEPTOR
end signal
end module
```

Figura 3: Módulo PROTOCOLO

- El *timeout* es de 1 s.
- El tiempo para procesar un paquete de información o un acuse de recibo es de 13,5 ms.

Para incluir estos conceptos de comportamiento en una especificación funcional ESTEREL, consideramos dos tipos de información cuantitativa:

El tiempo de tratamiento estimado para cada *acción temporizada*. Esta es una acción que consume tiempo, por ejemplo el tiempo para procesar un paquete, el tiempo de transmisión...: son acciones propias de la modelización del comportamiento pero que no tienen sentido en un modelo reactivo *Síncrono*. Debemos evitar en este punto cualquier incongruencia con respecto a la hipótesis de sincronía perfecta.

La probabilidad asociada a cada *comportamiento alternativo* que se pueda presentar. Cuando una probabilidad define el comportamiento de un sistema, cierto grado de no determinismo se introduce en el modelo reactivo sí}ncrono, determinista por naturaleza. Una vez más, las decisiones de modelización deben tratar con cuidado este aspecto.

Para incluir esas informaciones, se completa el modelo reactivo síncrono con dos tipos de *constructores de comportamiento*:

El constructor de acciones temporizadas asume la existencia de una señal global específica que representa el tiempo, **TS**. La unidad de tiempo que representa depende de la abstracción del modelo. Se trata de una entrada desde el punto de vista de cada módulo reactivo. **await n TS** determina el tiempo asociado a la acción temporizada corriente, la cual es definida mediante dos eventos: uno inicial y uno final. Así p.ej. la construcción de un paquete de información en el proceso de emisión, modelizada mediante la instrucción **emit** ENVIA_DATO en el módulo reactivo EMISOR pasa a ser:

```
emit ENVIA_DATO_I;
await 10 TS;
emit ENVIA_DATO_F;
```

donde cada ocurrencia del evento TS corresponde a 0.1 ms; el lapso de 1 ms para esta acción es así tomado en cuenta. Según esta misma convención, el *timeout* y el tiempo de transmisión (respectivamente señales TIMEOUT y TIEMPO_DE_TRANSMISION en el modelo de la **sección 3**) se modelizan con **await** 10000 TS y **await** 1067 TS respectivamente.

El constructor para comportamientos alternativos usa una función implementada externamente en **C**, PROBABILIDAD ***(valor)***, que devuelve ***true*** con la probabilidad ***valor***. El

```
module EMISOR
input RECIBE_ACK,TIMEOUT;
outputRECIBE,ENVIA_DATO

loop
 emitRECIBE
 emitENVIA_DATO
 do
      loop
        awaittTIMEOUT
        emitENVIA_DATO;
      end loop;
 watcing inmediate RECIBE_ACK;
 end;
end loop
end module
```

```
moduleRECEPTOR
inputRECIBE_DATO;
outputENTREGA,
          ENVIA_ACK;
loop
 awaitRECIBE_DATO;
 emitENTREGA;
 emit ENVIAACK;
endloop
end module
```

Figura 4: Módulos EMISOR y RECEPTOR

módulo MEDIUM de la **figura 6** es un ejemplo para ilustrar esta función y declara que la emisión de la señal RECIBE_DATO se hará con una probabilidad del 95%, lo que corresponde a una pérdida del 5% de los paquetes de información.

5. Instrumentación del Modelo

El trabajo complementario que debe realizarse para poder obtener medidas sobre el comportamiento del sistema es la instrumentación de los modelos con mecanismos de monitorización apropiados. Con este fin proponemos incluir un nuevo módulo, el MONITOR de la **figura 7**, encargado de observar un conjunto de señales definidas por el diseñador. Es responsabilidad de este último el decidir cuales son los eventos que necesita para calcular las medidas de comportamiento deseadas. El monitor utiliza una función auxiliar, GUARDA_ENTRADA(*<tiempo>*,*<señal>*), para memorizar todas las ocurrencias de cada señal que desee observar. Esta función escribe la fecha de la ocurrencia y el nombre de la señal correspondiente en un fichero de resultados. El mecanismo de monitorización propuesto ofrece únicamente la funcionalidad básica de observación necesaria para la medida del comportamiento. Una herramienta estadística completa debe incluir por lo menos el cálculo de los intervalos de confianza, necesarios para el control del tiempo de simulación.

6. Estudio preliminar

Analizamos en una primera etapa la posibilidad de instrumentar modelos ESTEREL con los constructores de comportamiento propuestos y con el mecanismo de monitorización para su uso como técnica de modelización orientada por la evaluación del comportamiento. Esta primera aproximación fue realizada a partir del modelo del protocolo escrito en LOTOS estocástico. LOTOS ofrece reglas para calcular un modelo dinámico a partir de una especificación dada; este modelo puede representarse mediante un autómata. Con LOTOS estocástico, Rico y Bochmann (10) modificaron algunas de estas reglas para poder introducir en las especificaciones las probabilidades y la duración de las acciones. Estas reglas modificadas calculan las etiquetas de comportamiento que son añadidas en los autómatas. La **figura 8** es el resultado de aplicar estas nuevas reglas en la especificación del protocolo *Stop & Wait*.

```
module MEDIUM:
input ENVIA_DATO,ENVIA_ACK,ERROR,TIEMPO_DE_TRANSMISION
output RECIBE_DATO_, RECIBE_ACK
loop
      await
          case inmediate ENVIA_DATOS do
              do
                    await ERROR
              watching TIEMPO_DE_TRANSMISION
              timeout
                    emit RECIBE_DATO
              end
          case inmediate ENVIA_ACK do
              do
                    await ERROR
              watching TIEMPO_DE_TRANSMISION
              timeout
                    emit RECIBE_ACK
              end
      end await
end loop
end module
```

Figua 5: Módulo MEDIUM

```
module MEDIUM:
FUNCTION probabilidad (integer): boolean:
...     if PROBABILIDAD (95)
        then [
                emit RECIBE_DATO_i;
                await 135 TS;
                emit RECIBE_DATO_F; ]
        end ...
end module
```

Figura 6: Módulo MEDIUM con el constructor de comportamientos alternativos

Un modelo ESTEREL sencillo puede obtenerse a partir de este autómata, definiendo un módulo reactivo para cada estado y una señal para cada transición. El modelo incluye la información cuantitativa de comportamiento mediante el uso de los constructores propuestos. P. ej., el módulo asociado al tercer estado del autómata de la figura 8 se presenta en la **figura 9**.

Para permitir el cálculo del *throughput* a partir de la simulación, únicamente necesitamos observar las ocurrencias de la señal **S5**, correspondientes al evento ENTREGA en la abstracción inicial. El código resultante del módulo principal, que incluye la presencia de un monitor, se detalla en la **figura 10**. Observemos que una señal inicial INIT debe emitirse para lanzar la simulación, ya que el módulo REACTIVO_1 está esperando por ella. La simulación fue implementada en el entorno de AGEL. Esta herramienta ofrece un temporizador que puede asociarse a cualquier señal de entrada, y que hemos 'conectado' a la entrada TS. Una vez la simulación interrumpida, lo que puede hacerse en cualquier momento, se calculan las medidas de comportamiento deseadas, en este caso el *throughput*, y posteriormente continuar con la simulación si no hemos alcanzado una estabilidad satisfactoria. Este procedimiento puede realizarse gracias a las posibilidades del simulador.

El *throughput* obtenido es de 2,819 paquetes por segundo, el cual se acerca a los valores calculados (2,852) con cadenas de Markov en LOTOS estocástico (10) o los calculados (2,853) con métodos analíticos en las redes de colas derivadas (13). Igualmente se aproxima a los 2,75 paquetes por segundo que se obtienen con redes de Petri estocásticas y distribuciones exponenciales (7).

Por consiguiente, este estudio permite validar la metodología de instrumentación propuesta para modelizar el comportamiento de protocolos con modelos Esterel adaptados.

7. Modelo del Comportamiento

En esta sección presentamos el modelo de comportamiento diseñado a partir de la especificación del protocolo *Stop & Wait* de la sección 3. Un nivel representando el punto de vista

```
module MONITOR:
procedure GUARDA_ENTRADA () (integer, string);
input TS, RECIBE, ENTREGA;
var TIEMPO : = 0 : integer in
loop
      await TS
      TIEMPO: TIEMPO + 1;
      present RECIBE then call GUARDA_ENTRADA ()
                          (TIEMPO; "RECIBE") ; end;
      present ENTREGA then call GUARDA_ENTRADA ()
                          (TIEMPO; "ENTREGA") ; end;
end loop
end var
end module
```

Figura 7: Módulo MONITOR

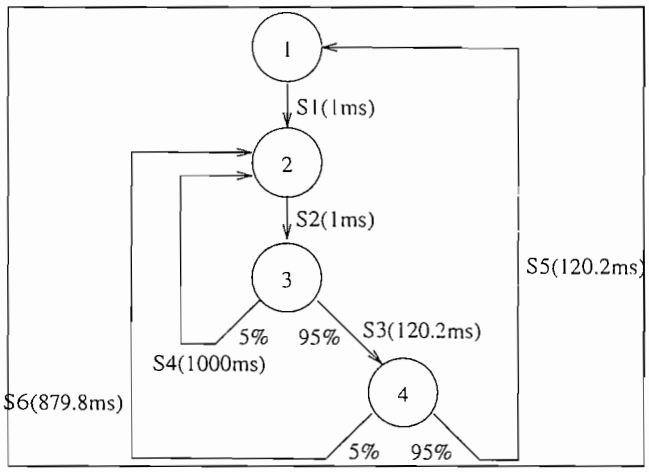


Figura 8: Autómata del protocolo

del usuario del protocolo ha sido incorporado con el fin de incluir las comunicaciones entre las entidades del protocolo y los usuarios. El módulo principal es ahora el reactivo SISTEMA de la **figura 11**.

El módulo ESTACION_A (resp. ESTACION_B) es un bucle sencillo a la espera de las señales RECIBE_I y RECIBE_F provenientes de la entidad emisora (resp. a la espera de la señal ENTREGA_ originada por el receptor), ante lo cual emite una señal RECIBE (resp. ENTREGA) hacia el entorno. El reactivo PROTOCOLO difiere del presentado en la figura 3 únicamente en la declaración de su interfaz: TS es ahora la única entrada, las salidas son RECIBE_I, RECIBE_F y ENTREGA_, y el conjunto de señales locales está definido por ENVIA_ACK_, ENVIA_DATO_I, ENVIA_DATO_F, RECIBE_ACK_I, RECIBE_ACK_F, RECIBE_DATO_I y RECIBE_DATO_F.

Los módulos reactivos EMISOR y RECEPTOR incluyen ahora varios constructores de acciones temporizadas, siendo los de las **figuras 12 y 13.** Observemos la presencia del constructor de comportamientos alternativos en el módulo MEDIUM, que ahora se define como lo muestra la **figura 14.** Finalmente, el módulo MONITOR es el mismo que el de la figura 7. Las señales RECIBE y ENTREGA son observadas de esta manera, permitiendo el cálculo del *throughput* durante la simulación.

Para nuestro estudio del protocolo, obtuvimos valores muy similares a los de referencia. El *throughput* medido es de 2,858 paquetes por segundo, que puede compararse con nuestro primer resultado 2,819 y con el resultado 2,859 de la simulación con redes de colas derivadas de LOTOS (13).

```
module REACTIVO_3:
function PROBABILIDAD (integer): boolean;
input S2, TS;
output s3, s4;
loop
        await inmediate S2;
        if PROBABILIDAD (5)
                then [ await 100000 TS;
                        emit S4; ]
                else [ await 1202 TS;
                        emit S3; ]
        end if
end loop
end module
```

Figura 9: Uno de los reactivos para el estudio preliminar

8. Conclusiones

El desarrollo de sistemas informáticos que incluyan fuertes exigencias en términos de tiempo (sistemas tiempo real y acoplados, algunos sistemas de tipo conversacional...), traducidas básicamente en tiempos de respuesta, debe realizarse siguiendo criterios cuantitativos adicionales, que se toman en cuenta mediante el uso de modelos de comportamiento. Estos deben coexistir con los demás modelos de diseño, sea porque no cubren todos los aspectos funcionales del diseño o porque otros criterios no funcionales (seguridad, robustez, etc.) se consideran igualmente importantes. Nos interesamos en la información necesaria para construir modelos de comportamiento, identificando la que podemos extraer de otros modelos de diseño, de los modelos reactivos síncronos en particular. El proceso de modelización para la evaluación del comportamiento puede facilitarse si se crean los mecanismos necesarios a la reutilización de los aspectos funcionales del diseño.

Este trabajo muestra cómo podemos obtener resultados cuantitativos sobre el comportamiento del sistema a partir de modelos reactivos síncronos *instrumentados*. Proponemos una metodología de instrumentación en dos etapas:

- ***Incluir en el modelo la información cuantitativa necesaria para la evaluación del comportamiento;*** para ello proponemos dos constructores, un constructor para definir acciones temporizadas que asume un referencial de tiempo global, y un constructor para comportamientos alternativos que toma en cuenta sus probabilidades respectivas.
- ***Monitorizar el modelo con un módulo reactivo dedicado,*** el monitor, encargado de determinar en cada evento de la simulación la presencia o la ausencia de ciertas señales predefinidas. Escogiendo las señales convenientemente, la información producida por el monitor durante la simulación permite calcular las medidas de comportamiento deseadas.

Aplicamos esta metodología para el estudio del caso de un protocolo, validando los modelos obtenidos con ESTEREL como lenguaje de modelización y AGEL como herramienta de desarrollo y soporte para la simulación. Nuestra experiencia en el uso de esta estrategia de integración debe permitirnos profundizar dos aspectos:

- la generalización de un conjunto de reglas de instrumentación con el fin de asegurar la cobertura de la semántica específica de ESTEREL;
- la adecuación del entorno de simulación para ofrecer más posibilidades de con-trol de simulación y cálculo de resultados de comportamiento.

```
module TODOS_LOS_REACTIVOS:
input TS;
signal INIT, S1, S2, S3, S4, S5, S6, in
                emit INIT;
        ||
                run REACTIVO 1
        ||
                run REACTIVO2
        ||
                run REACTIVO 3
        ||
                run REACTIVO 4
        ||
                run REACTIVO_MONITOR
end signal
end module
```

Figura 10: Reactivo principal para el estudio preliminar

```
module SISTEMA:
input TS;
output RECIBE, ENTREGA;
signal RECIBE_I, RECIBE_F, ENTREGA_
in
        run ESTACION_A
        ||
        run PROTOCOLO
        ||
        run MONITOR
        ||
        run ESTACION_B
end signal
end module
```

Figura 11: Reactivo principal del modelo del comportamiento

Los modelos de éste pueden obtenerse directamente a partir de especificaciones formales. El ingeniero de desarrollo debe así poder elegir entre las diferentes opciones de diseño de acuerdo con las estimaciones sobre el comportamiento de las que puede disponer desde las primeras etapas del ciclo de vida, como permite establecer la ingeniería del comportamiento.

Referencias

(1) Agel. *Workshop Manual, Version 3.0.* Ilog s.a., 2 avenue Galliéni, 94253 Gentilly.

(2) Berry,G.; *The Semantics of Pure Esterel.* In Program Design Calculi, Broy, M. (ed.), NATO ASI Series, Computer and System Sciences 118 (1993) 361-409.

(3) Berry,G.; Gonthier,G.: *The Esterel Synchronous Programming Language: Design, Semantics, Implementation.* Science Of Computer Programming 19, 2 (1992) 87-152.

(4) Conquet,E.; Valderruten,A.; Trémoulet,R.; Raynaud,Y.; Ayache, S.: *Un modèle du processus de l'activité d'évaluation des performances.* Génie Logiciel et Systèmes Experts 27 (1992) 27-31.

(5) Esterel V3; *Language Reference Manual.* CISI Ingénierie, Les Cardoulines B1, 06560 Valbonne, France.

(6) *Standard Glossary of Software Engineering Terminology.* IEEE 610.12-90 (1990).

(7) Molloy,M.K.; *Performance Analysis using Stochastic Petri Nets.* Transactions on Computers 31, 9 (1982) 913-917

(8) ISO: *The Reference Model.* DIS 7498 Part 1/4 (1987).


```
module EMISOR:
input RECIBE_ACK_I, RECIBE_ACK_F, TS;
output RECIBE_I, RECIBE_F, ENVIA_DATO_I, ENVIA_DATO_F;
loop
        emit RECIBE_I;
        await 10 TS;                %lapso de 1 ms
        emit RECIBE_F;
        emit ENVIA_DATO_I;
        await 10 TS;                % lapso de 1 ms
        emit ENVIA_DATO_F;;
        do
                loop
                        await 100000 TS;% Timeout: 1 s
                        emit ENVIA_DATO_I;
                        await 10 TS;
                        emit ENVIA_DATO_F;
                end loop;
        watching inmediate RECIBE_ACK_I
        timeout
                await 135 TS;  % Proceso de un paquete: 13.5 ms
                await inmediate RECIBE_ACK_F;
        end;
end loop
end module
```

Figura 12: Reactivo emisor del modelo de comportamiento

```
module RECEPTOR:
input RECIBE_DATO_I, RECIBE_DATO_F, TS;
output ENVIA_ACK_, ENTREGA_;
loop
        await RECIBE_DATO_I;
        await 135 TS;           % Proceso de un paquete: 13.5 ms
        await inmediate RECIBE_DATO_F;
        emit ENTREGA_;
        emit EMVIA_ACK_;        % sin lapso de tiempo
end loop
end module
```

Figura 13: Reactivo receptor del modelo del comportamiento

(9) Razouk,R.; Phelps,C.; *Performance Analysis using Timed Petri Nets.* In Protocol Specification, Testing and Verification IV, IFIP (1984) 561-576.

(10) Rico,N.; Bochmann, G.v.; *Performance Description and Analysis for Distributed Systems using a variant of LOTOS.* In Protocol Specification, Testing and Validation X, IFIP (1990).

(11) Schwartz,R.L.; Melliar-Smith,P.M.; Vogt,F.H.; *Interval Logic: A Higher-Level Temporal Logic for Protocol Specification.* In Protocol Specification, Testing and Verification III, Rudin, H., West, C.H. (eds.), IFIP (1983).

(12) Valderruten, A.; *Modélisation des Performances et Développement de Systèmes Informatiques: une étude d'intégration.* Thèse d'Informatique, Université Paul Sabatier, Toulouse (1993).

(13) Valderruten,A.; Hjiej,O. Benzekri,A.; Gazal,D.: *Deriving Queueing Networks Performance Models from Annotated LOTOS Specifications.* In Computer Performance Evaluation'92: Modelling Techniques and Tools, Pooley, R., Hillston, J. (eds.), Edinburgh University Press (1993) 120-130.

(14) Valderruten,A.; Vilares,M.; Graña,J.: *Instrumentation of Synchronous Reactive Models for Performance Engineering,* Lecture Notes in Computer Science 989: Software Engineering - ESEC'95 (pp.76-89), Springer-Verlag, 1995.


```
module MEDIUM:
function PROBABILIDAD (INTEGER):boolean;
input ENVIA_DATO_I, ENVIA_DATO_F, ENVIA_ACK_, TS;
output RECIBE_DATO_I, RECIBE_DATO_F,
RECIBE ACK_I, RECIBE_ACK_F;
loop
  await
   case inmediate ENVIA_DATO_I do
        await 10 TS;            % lapso de 1 ms
        await inmediate ENVIA_DATO_F;
        await 1067 TS;          % Tiempo de transmision 106.7 ms
        if PROBABILIDAD (95)  %Probabilidad de perdida: 5%
        then [
                emit RECIBE_DATO_I;
                await 135 TS;
                emit RECIBE_DATO_F; ]
        end
    case inmediate ENVIA_ACK_ do
        await 1067 TS;          %Tiempo de transmision
        if PROBABILIDAD (95)  %Probabilidad de perdida:5%
        then [
                emit RECIBE_ACK_I;
                await 135 TS;
                emit RECIBE_ACK_F; ]
        end
    end await
end loop
end module
```

Figura 14: Reactivo MEDIUM del modelo de comportamiento

Ingeniería del Software

Procesos del ciclo de vida del software

José Luis Esteban, Mario G. Piattini
Miembros de AENOR 71/SC7

Este artículo presenta una de las normas internacionales más interesantes para la ingeniería del software, titulada «*Software life-cycle processes*» (Procesos del ciclo de vida del software), aprobada como norma internacional ISO/IEC IS 12207-1 (draft) en 1995, de la que hemos terminado su traducción para publicarla.

1. Introducción

El problema de definir una marco de referencia común, que pueda ser empleado por todas las personas que participan en un desarrollo informático, en el que se definan los procesos, actividades y tareas a desarrollar es uno de los más importantes en cualquier departamento de sistemas de información. A lo largo del tiempo se han propuesto distintos paradigmas o ciclos de vida para el software (aunque en la normativa oficial se use el término 'soporte lógico' para referirse al software, empleamos aquí este último por ser más común): desde el ciclo en cascada, pasando por el modelo en espiral de Boehm, hasta los más recientes ciclos de vida para sistemas orientados al objeto, véase PIATTINI (1995). Novática ha prestado atención a este tema en repetidas ocasiones, destacando, p.ej. COSTA (1989).

Afortunadamente también los organismos internacionales se han venido ocupando del ciclo de vida del software y, después de varios años de trabajo, ISO/IEC han publicado la norma internacional IS 12207-1 (draft). Esta norma entiende por modelo de ciclo de vida: «*Un marco de referencia que contiene los procesos, actividades y tareas involucradas en el desarrollo, operación y mantenimiento de un producto de software, abarcando la vida del sistema desde la definición de los requisitos hasta la finalización de su uso*»; considerando una actividad como un conjunto de tareas y una tarea como una acción que transforma entradas en salidas. Este marco consta de los principales procesos que pueden aplicarse para adquirir, suministrar, desarrollar, operar y mantener el software. También incluye un proceso que puede ser empleado para controlar y mejorar las disciplinas de su gestión e ingeniería.

La norma agrupa las actividades que pueden ser realizadas durante el ciclo de vida del software en 5 procesos principales, 8 procesos de apoyo y 4 procesos generales, así como el proceso de adaptación a casos concretos (**figura 1**); todo lo cual presentamos de forma resumida en los próximos epígrafes.

2. Presentación de la norma

Lo primero que debemos señalar es que la norma no fomenta o especifica ningún modelo concreto de ciclo de vida, gestión del software o método de ingeniería, ni prescribe cómo realizar ninguna actividad. La norma presenta un marco de referencia para el proceso del ciclo de vida del software, estructurado en cinco procesos 'principales':

- **Proceso de adquisición:** define las actividades de la organización que adquiere un sistema o producto de software.
- **Proceso de suministro:** define las actividades de la organización que proporciona el producto de software.
- **Proceso de desarrollo:** define las actividades de la organización que define y desarrolla el producto software.
- **Proceso de operación:** define las actividades de la organización que proporciona el servicio de operación de un sistema informático en su entorno y para sus usuarios.
- **Proceso de mantenimiento:** define las actividades de la organización que proporciona el mantenimiento del software; esto es, la gestión de las modificaciones necesarias para mantener el software actualizado y operativo, incluyendo la migración y la retirada.

Con estos cinco se cubre todo el ciclo de vida del software, desde la materialización de su necesidad hasta su retirada operativa. Sin embargo, la realización de estos procesos resulta complicada, por lo que se han definido otros dos grupos de procesos, de apoyo y organizacionales, que ahora se definen, con el fin de facilitar y detallar la realización comentada.

Grupos de procesos de 'apoyo'

- **Proceso de documentación:** define las actividades para registrar la información producida por un proceso del ciclo de vida.
- **Proceso de gestión de la configuración:** define las actividades de gestión de la configuración.
- **Proceso de aseguramiento de la calidad:** define las actividades para asegurar objetivamente que los productos de software se ajustan a la especificación de requisitos y plan establecido.
- **Proceso de verificación:** define las actividades para verificar los productos de software con distinta profundidad, dependiendo del proyecto de software.
- **Proceso de validación:** define las actividades para la validación de los productos de software.
- **Proceso de revisión conjunta:** define las actiivdades para evaluar el estado y los productos de una actividad.
- **Proceso de auditoría:** define las actividades para determinar el grado en que se cumplen los requisitos, planes y contrato.

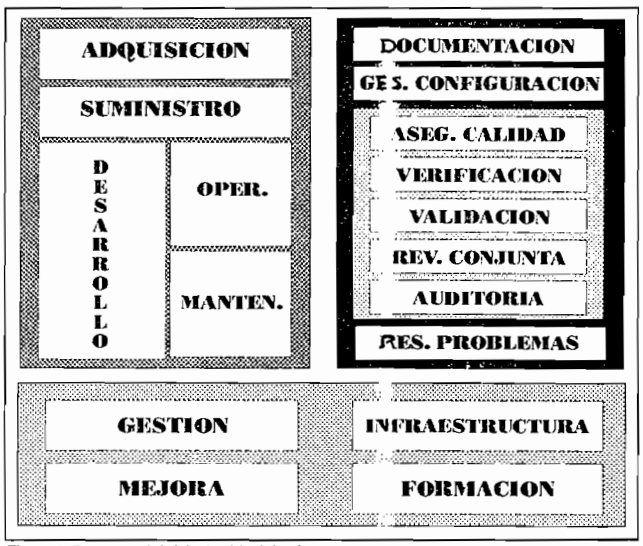


Figura 1: Procesos del ciclo de vida del software

- **Proceso de resolución de problemas:** define un proceso para analizar y eliminar los problemas, cualquiera que sea su naturaleza u origen, descubiertos durante el desarrollo, operación, mantenimiento u otros procesos.

Grupo de procesos 'organizacionales'
- **Proceso de gestión:** define las actividades básicas de la gestión, incluyendo la gestión del proyecto.
- **Proceso de infraestructura:** define las actividades básicas para establecer la estructura fundamental de un proceso.
- **Proceso de mejora:** define las actividades básicas que una organización ejecuta para establecer, medir, controlar y mejorar los procesos del ciclo de vida.
- **Proceso de formación:** define las actividades para proporcionar una formación adecuada al personal.

En resumen, esta visión general ha presentado: un grupo de procesos 'principales', que estructura y permite cubrir todo el ciclo de vida del software; otro grupo de procesos de apoyo que ayudan en su realización; y otro grupo de procesos organizacionales que ayudan a gestionar el ciclo de vida del software. Importa señalar que los procesos de apoyo también ayudan en la realización de los procesos organizacionales y éstos a su vez permiten gestionar la realización de los procesos de ayuda.

3. Procesos principales

Antes de detallar todos los procesos y las actividades que los componen, se señala que en la norma, los términos 'organización' y 'parte' son casi sinónimos.Una organización es un grupo de personas organizadas con algún propósito específico, por ejemplo, una empresa. Cuando una organización firma un contrato se convierte en una parte. Las partes pueden ser de la misma o de diferentes organizaciones. La norma considera 'parte principal' la que inicia o realiza el desarrollo, operación o mantenimiento del software. Las partes principales son el 'adquiriente' (comprador, cliente, usuario o propietario), el suministrador, el desarrollador, el operador y el personal de mantenimiento del software. Los procesos principales son:

3.1. Proceso de adquisición

El proceso comienza definiendo la necesidad de adquirir un sistema o un producto ('producto' también incluye 'servicio') software; sigue con la preparación y publicación de la solicitud de propuestas, selección de un suministrador y gestión de los procesos de adquisición, hasta la aceptación del producto. El adquiriente gestiona este proceso a nivel de proyecto.
Lista de actividades (5): Iniciación; Preparación de la petición de propuestas; Preparación y actualización del contrato; Seguimiento del suministro; Aceptación y terminación.

3.2. Proceso de suministro

Este proceso puede iniciarse ya sea por una decisión de preparar una propuesta para responder a una petición de un adquiriente, o por la firma de un contrato con el adquiriente para proporcionar el producto software.
Lista de actividades (7): Iniciación; Preparación de la respuesta, Contrato; Planificación; Ejecución y control; Revisión y evaluación; Entrega y terminación.

3.3. Proceso de desarrollo

Contiene las actividades para el análisis de requisitos, diseño, codificación, integración, pruebas, e instalación y aceptación relativas al software. Puede contener actividades relativas al sistema si están estipuladas en el contrato.
Lista de actividades (13): Implementación del proceso; Análisis de requisitos del sistema; Diseño de la arquitectura del sistema; Análisis de requisitos del software; Diseño de la arquitectura del software; Diseño detallado del software; Codificación y prueba del software; Integración del software; Prueba de cualificación del software; Integración del sistema; Prueba de idoneidad del sistema; Instalación del software; Soporte a la aceptación del software.

Debido al interés que tiene este proceso, resumimos algunas tareas que la norma especifica para cada actividad.

3.3.1. Implementación del proceso

Si no se estipula en el contrato, el desarrollador definirá o seleccionará un modelo de ciclo de vida del software apropiado al enfoque, magnitud, y complejidad del proyecto. El desarrollador realizará las siguientes tareas:
- Documentar las salidas;
- Incluir las salidas en el proceso de gestión de la configuración;
- Documentar y resolver los problemas y disconformidades de acuerdo con el proceso de resolución de problemas;
- Otros procesos de apoyo según especificación del contrato.

El desarrollador seleccionará, adaptará y empleará normas internas, metodologías, procedimientos, y lenguajes de programación (si no se estipulan en el contrato) que estén documentados, adecuados, y establecidos por la organización para realizar el proceso de desarrollo y los procesos de apoyo.

El desarrollador elaborará planes para gestionar las actividades del proceso de desarrollo, que incluirán normas especificas, métodos, herramientas, acciones y la responsabilidad asociada con el desarrollo y cualificación de todos los requisitos incluyendo la prevención (*safety*) y la seguridad (*security*).

Se pueden emplear productos no entregables en el desarrollo o mantenimiento del software. Sin embargo, debe asegurarse que la operación y el mantenimiento del software entregable después de su entrega al comprador es independiente de tales productos, en caso contrario, también éstos deben entregarse.

3.3.2. Análisis de requisitos del sistema

- Se especificarán los requisitos del sistema que deben incluir: sus funciones y capacidades, prevención, seguridad, ingeniería humana, interfaces, operaciones y requisitos de mantenimiento, restricciones de diseño y requisitos de cualificación.
- Se evaluarán los requisitos del sistema considerando criterios como la trazabilidad, consistencia, verificabilidad o viabilidad.

3.3.3. Diseño de la arquitectura del sistema

Deberá establecerse el nivel superior de la arquitectura del sistema que identificará los elementos de hardware y software (que se identificarán como elementos de la configuración), así como las operaciones manuales.

La arquitectura del sistema y los requisitos para los elementos de la configuración y operaciones manuales se evaluarán considerando los criterios citados de trazabilidad, consistencia, verificabilidad o viabilidad.

3.3.4. Análisis de los requisitos del software

El desarrollador:
- establecerá y documentará los requisitos del software, incluyendo la especificación de las características de calidad.
- evaluará los requisitos del software.
- llevará a cabo las reuniones de revisión y establecerá una línea base para los requisitos de los elementos de la configuración una vez pasadas las revisiones con éxito.

3.3.5. Diseño de la arquitectura del software

El desarrollador
- transformará los requisitos para los elementos de la configuración en una arquitectura que describa su estructura de alto nivel e identifique los componentes principales.
- realizará y documentará un diseño a alto nivel para la interfaz externa con los elementos de la configuración y entre éstos.
- realizará y documentará un diseño a alto nivel para la base de datos.
- deberá elaborar y documentar la versión preliminar de los manuales de usuario.
- definirá y documentará los requisitos de las pruebas preliminares y la planificación para la integración del software.
- evaluará la arquitectura de los elementos de la configuración y los diseños de la interfaz y la base de datos.
- llevará a cabo revisiones conjuntas.

3.3.6. Diseño detallado del software

El desarrollador:
- realizará un diseño detallado para cada componente software de los elementos de la configuración (los componentes software serán refinados suficientemente en niveles inferiores conteniendo unidades de software que puedan ser codificadas, compiladas y probadas).
- realizará y documentará un diseño detallado para las interfaces externas con los elementos de la configu-ración, entre los componentes y entre las unidades de software.
- realizará y documentará un diseño detallado para la BD.
- actualizará los manuales de usuario.
- definirá y documentará los requisitos de prueba y planificará las unidades del software para su prueba (los requisitos de prueba deberían incluir las pruebas de sobrecarga -*stress*- de las unidades de software hasta los límites de sus requisitos).
- actualizará los requisitos de prueba y la planificación para la integración del software.
- evaluará el diseño detallado del software y los requisitos de prueba.
- llevará a cabo reuniones de revisión.

3.3.7. Codificación del software y pruebas

El desarrollador:
- realizará y documentará cada unidad de software y la base de datos;
- realizará y documentará los procedimientos de prueba y los datos para probar cada unidad de software y la base de datos.
- probará cada unidad de software, y la base de datos asegurando que satisface los requisitos.
- actualizará los manuales de usuario.
- actualizará los requisitos de prueba y la planificación para la integración del software.
- evaluará el código y los resultados de las pruebas del software.

3.3.8. Integración del software

El desarrollador:
- elaborará un plan para integrar las unidades y los componentes software en los elementos de la configuración (el plan incluirá los requisitos de prueba, procedimientos, datos, responsabilidades y planificación).
- integrará las unidades y los componentes software junto con las pruebas.
- actualizará los manuales de usuario.
- llevará a cabo y documentará, para cada cualificación de requisitos de los elementos de la configuración, un conjunto completo de pruebas, casos de pruebas (entradas, salidas, criterios), y procedimientos de prueba para dirigir la prueba de cualificación del software.
- evaluará el plan de integración, diseño, codificación, pruebas, resultados de las mismas, y los manuales de usuario.
- llevará a cabo reuniones de revisión.

3.3.9.- Prueba de cualificacion del software

El desarrollador:
- llevará a cabo la prueba de cualificación de acuerdo con los requisitos especificados.
- actualizará los manuales de usuario.
- evaluará el diseño, codificación, pruebas, resultados de la prueba, y manuales de usuario.
- dará apoyo a la(s) auditoría(s).

Una vez que las auditorías finalicen con éxito, el desarrollador:
- deberá actualizar y preparar el software entregable para la integración del sistema, prueba de cualificación del mismo, instalación del software, y aceptación del software;
- deberá establecer una línea base para el diseño y el código de los elementos de la configuración.

3.3.10. Integración de sistemas

- Se integrarán en el sistema los elementos de la configuración software y hardware, operaciones manuales y otros sistemas.
- Para cada cualificación de requisitos del sistema se desarrollarán y documentarán un conjunto completo de pruebas, casos de prueba (entradas, salidas, criterios) y procedimientos de prueba.
- Se evaluará el sistema integrado.

3.3.11. Prueba de cualificación del sistema

- La prueba de cualificación del sistema se llevará a cabo de acuerdo con los requisitos de cualificación especificados para el sistema.
- Se evaluará el sistema.
- El desarrollador apoyará la(s) auditoría(s).

Una vez que la(s) auditoría(s) finalicen con éxito, si se han realizado, el desarrollador deberá:
- actualizar y preparar los elementos de la configuración entregables para la instalación del software y la ayuda a la aceptación del software.
- establecer una línea base para el diseño y el código de los elementos de la configuración.

3.3.12. Instalación del software

El desarrollador:

- elaborará un plan para instalar el software en el entorno de destino.
- deberá instalar el software de acuerdo con el plan de instalación.

3.3.13. Ayuda a la aceptación del software

El desarrollador:
- ayudará a la revisión de aceptación y prueba que realice el adquiriente del software.
- completará y entregará la documentación y el código del software.
- proporcionará una formación inicial y continuada, y apoyará al adquiriente según se especifique en el contrato.

3.4. Proceso de operación

Este proceso abarca la operación del software y el soporte operacional a usuarios. Debido a que la operación del software se integra en la operación del sistema, las actividades y tareas del proceso de operación se refieren al sistema.
Lista de actividades (4): Implementación del proceso; Prueba operativa; Operación del sistema; Soporte a usuarios.

3.5. Proceso de mantenimiento

Este proceso se activa cuando el software sufre modificaciones de código o de documentación asociada debido a un error, una deficiencia, un problema o la necesidad de mejora/adaptación. El objetivo es modificar un software existente manteniendo su integridad. Este proceso incluye la migración y retirada del software y termina con la retirada del servicio del software.
Lista de actividades (6): Implementación del proceso; Análisis del problema y de la modificación; Implementación de la modificación; Revisión/aceptación del mantenimiento; Migración; Retirada del software.

4. Procesos de apoyo

Los procesos de apoyo se emplean en varios puntos del ciclo de vida y pueden ser realizados por la organización que los emplea, por una independiente (como un servicio), o por un cliente como elemento planificado o acordado del proyecto.

4.1. Proceso de documentación

El proceso de documentación registra la información producida por un proceso o actividad del ciclo de vida.
Lista de actividades (4): Implementación del proceso, Diseño y desarrollo, Producción, Mantenimiento.

Aquí se señala también que todo documento debe incluir:
- Título o nombre,
- Propósito,
- Destinatario,
- Procedimientos y responsabilidades para entrada, desarrollo, revisión, modificación, aprobación, producción, almacenamiento, distribución, mantenimiento y gestión de la configuración;
- Planificación para versiones intermedias y finales.

4.2. Proceso de gestión de la configuración

La gestión de la configuración es un proceso de aplicación de procedimientos administrativos y técnicos durante todo el ciclo de vida del sistema para:
- identificar, definir y establecer la línea base de los elementos de la configuración del software de un sistema,
- controlar las modificaciones y liberaciones de los elementos,
- registrar e informar sobre el estado de los elementos y las peticiones de modificación,
- asegurar la completitud, consistencia y corrección de los elementos,
- controlar su almacenamiento, manipulación y entrega.

Lista de actividades (6): Implementación del proceso, Identificación de la configuración, Control de la configuración, Contabilidad del estado de la configuración, Evaluación de la configuración, Gestión y entrega de la liberación (*release*).

4.3. Proceso de aseguramiento de la calidad

El proceso de aseguramiento de la calidad es un proceso que proporciona el aseguramiento adecuado de que los procesos y productos de soporte del ciclo de vida del proyecto cumplen con los requisitos especificados y se ajustan a los planes establecidos. El aseguramiento de la calidad puede utilizar los resultados de otros procesos de apoyo, como la verificación, la validación, las revisiones conjuntas, las auditorías y la resolución de problemas.
Lista de actividades (4): Implementación del proceso, Aseguramiento del producto, Aseguramiento del proceso, Aseguramiento de los sistemas de calidad.

4.4. Proceso de verificación

El proceso de verificación determina si los requisitos de un sistema o software son completos y correctos, y si los productos de software en cada fase de desarrollo cumplen los requisitos o condiciones impuestos sobre ellos en las fases previas.
Lista de actividades (2): Implementación del proceso, Verificación.

4.5. Proceso de validación

El proceso de validación es un proceso para determinar si los requisitos y el sistema o software construidos al final cumplen con su uso proyectado.
Lista de actividades (2): Implementación del proceso, Validación.

4.6. Proceso de revisión conjunta

El proceso de revisión conjunta es un proceso que sirve para evaluar el estado y los productos de software de una actividad del ciclo de vida o una fase de un proyecto. Las revisiones conjuntas se celebran tanto a nivel de gestión como técnico, teniendo lugar a lo largo de toda la vida del contrato. Este proceso puede emplearlo cualquiera de las dos partes, en el que una parte (parte revisora) revisa a la otra (parte revisada).
Lista de actividades (3): Implementación del proceso, Revisiones de dirección de proyecto, Revisiones técnicas.

4.7. Proceso de auditoría

El proceso de auditoría es un proceso para determinar el cumplimiento con los requisitos, planes y contrato según sea apropiado. Puede ser empleado por cualquiera de las dos partes, donde una parte (parte auditora) audita los productos de software o las actividades de la otra (parte auditada).
Lista de actividades (2): Implementación del proceso, Auditoría.

4.8. Proceso de resolución de problemas

El proceso de resolución de problemas es un proceso para analizar y eliminar los problemas (incluyendo disconformidades), cualquiera que sea su naturaleza o fuente, descubiertos durante el desarrollo, operación o mantenimiento u otros procesos. El objetivo es proporcionar un medio oportuno, responsable y documentado de asegurar que todos los problemas descubiertos se analizan y eliminan y que se identifican las tendencias.
Lista de actividades (2): Implementación del proceso, Resolución de problemas.

5. Procesos organizacionales

Estos procesos ayudan a establecer, implementar y mejorar una organización, consiguiendo que sea más efectiva. Se llevan a cabo normalmente a nivel organizacional (corporativo), fuera del ámbito de los proyectos y contratos específicos.

5.1. Proceso de gestión

El proceso de gestión contiene las actividades y tareas genéricas que puede emplear cualquier parte que tenga que dirigir su(s) respectivo(s) proceso(s).
Lista de actividades (5): Iniciación y definición del alcance, Planificación, Ejecución y control, Revisión y evaluación, Cierre.

5.2. Proceso de infraestructura

Este proceso establece la infraestructura necesaria para cualquier otro proceso, y puede incluir hardware, software, herramientas, técnicas, normas e instalaciones para desarrollo, operación o mantenimiento.
Lista de actividades (3): Implementación del proceso, Establecimiento de la infraestructura, Mantenimiento de la infraestructura.

5.3. Proceso de mejora

Este proceso permite establecer, valorar, medir, controlar y mejorar los procesos del ciclo de vida del software.
Lista de actividades (3): Establecimiento del proceso, Valoración del proceso, Mejora del proceso.

5.4. Proceso de formación

El proceso de formación es un proceso para proporcionar y mantener un personal formado.
Lista de actividades (3): Implementación del proceso, Desarrollo del material de formación, Implementación del plan de formación.

6. Proceso de adaptación

El proceso de adaptación es un proceso para realizar la adaptación básica de esta norma con respecto a los proyectos de software. Como sabemos, las variaciones en las políticas y procedimientos organizacionales, los métodos y estrategias de adquisición, el tamaño y complejidad de los proyectos, los requisitos de sistema y métodos de desarrollo, entre otros, influencian la forma de adquirir, desarrollar, operar o mantener un sistema. Esta norma se ha escrito para un proyecto general que incluye tantas variaciones como sea posible; por tanto, debe adaptarse para los proyectos concretos a fin de reducir costes y mejorar la calidad. El proceso consta de***4 actividades***:

6.1. Identificar entornos del proyecto

Esta actividad consta de la siguiente tarea: se deberán identificar las características del entorno del proyecto que influirán en la adaptación, tales como: modelos del ciclo de vida, fase actual del ciclo de vida del sistema actual, requisitos de sistema y software, políticas organizacionales, procedimientos y estrategias, tamaño, criticidad y tipos de sistema y software, número de personas y partes involucradas, etc.

6.2. Solicitar entradas (*inputs*)

Esta actividad consta de la siguiente tarea: se deberán solicitar las entradas de las organizaciones que se vean afectadas por las decisiones de adaptación. Se debería involucrar en la adaptación a los usuarios, personal de apoyo y de contratación, así como a los posibles ofertantes.

6.3. Seleccionar procesos, actividades y tareas

Esta actividad consta de las tres siguientes tareas:
- Se deberán decidir los procesos, actividades y tareas a realizar. Esto incluye la documentación a desarrollar y quién será responsable de la misma.
- Se deberán especificar en el propio contrato los procesos, actividades y tareas descritos anteriormente, pero no proporcionados en esta norma. Se deberían evaluar los procesos organizacionales para determinar si pueden proveer estos procesos, actividades y tareas.
- Se deberán considerar cuidadosamente las distintas tareas para decidir si se mantienen o eliminan en un proyecto o sector de negocio concreto, teniendo en cuenta el riesgo, coste, programa, rendimiento, tamaño, criticidad e interfaz humana.

6.4. Documentar decisiones de adaptación y sus causas

Esta actividad consta de una sola tarea: se deberán documentar todas las decisiones de adaptación junto con sus causas. La **figura 2** esquematiza un proceso de aplicación de la norma.

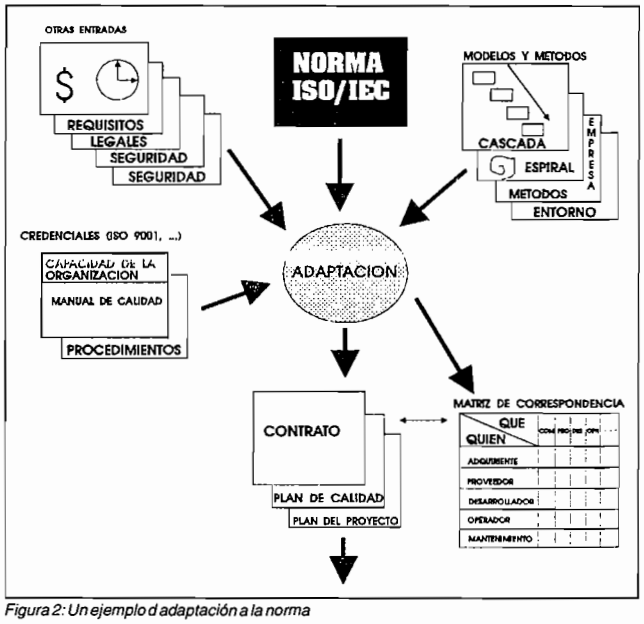


Figura 2: Un ejemplo d adaptación a la norma

7. Conclusiones

Hemos resumido la norma internacional ISO/IEC 12207-1 (DRAFT), que contiene los procesos que son aplicables a lo largo del ciclo de vida de un proyecto software. En la **figura 3** se muestran los procesos y sus interrelaciones bajo diferentes 'visiones' (puntos de vista) de utilización de la norma:

- **Contrato;** las partes adquiriente y proveedor negocian y firman un contrato, empleando los procesos de adquisición y suministro.
- **Gestión;** el adquiriente, el proveedor, el desarrollador, el operador y el personal de mantenimiento gestionan sus respectivos procesos para el proyecto de software.
- **Operación;** el operador proporciona el servicio de operación del software para los usuarios.
- **Ingeniería;** el desarrollador o el personal de mantenimiento llevan a cabo sus respectivas tareas de ingeniería para producir o modificar el producto de software.
- **Soporte;** las partes (tales como gestión de la configuración, aseguramiento de la calidad) proporcionan servicios de apoyo a otros en el cumplimiento de tareas únicas específicas.

Los procesos organizacionales se emplean a nivel corporativo por una organización con objeto de establecer e implementar una estructura subyacente compuesta de proceso(s) de ciclo de vida asociado(s) junto al personal y su mejora continua. Creemos que esta norma puede resultar muy interesante a las empresas e instituciones que quieran mejorar el proceso de desarrollo, ya sea por aumentar algún nivel en el modelo de capacidad de madurez (CMM) u obtener la certificación ISO 9000, puesto que establece un marco de referencia en el que se pueden insertar todos los elementos necesarios a tal fin: metodologías de desarrollo, gestión de proyectos, gestión de la configuración, aseguramiento de la calidad, formación, etc. Finalmente, aconsejamos la lectura detenida y completa de la norma (draft), y recordamos que esta presentación es un resumen subjetivo de la misma y en ningún caso la sustituye.

Agradecemos a todos los miembros del subcomité 7 del comité AENOR 71 las sugerencias que nos hicieron en la traducción de la norma y sus interesantes comentarios sobre la misma.

Bibliografía

COSTA, M. (1989); *El ciclo de vida del desarrollo de software.* En: Novática Vol. XIV, Nº 76, pp. 9-42.

ISO/IEC (1995); *Information Technology - Software - Part 1: Software life-cycle processes.* IS 12207-1.

PIATTINI, M. (1995); *Ciclos de vida para sistemas orientados al objeto.* En: Boletín del CUORE, julio 1995.

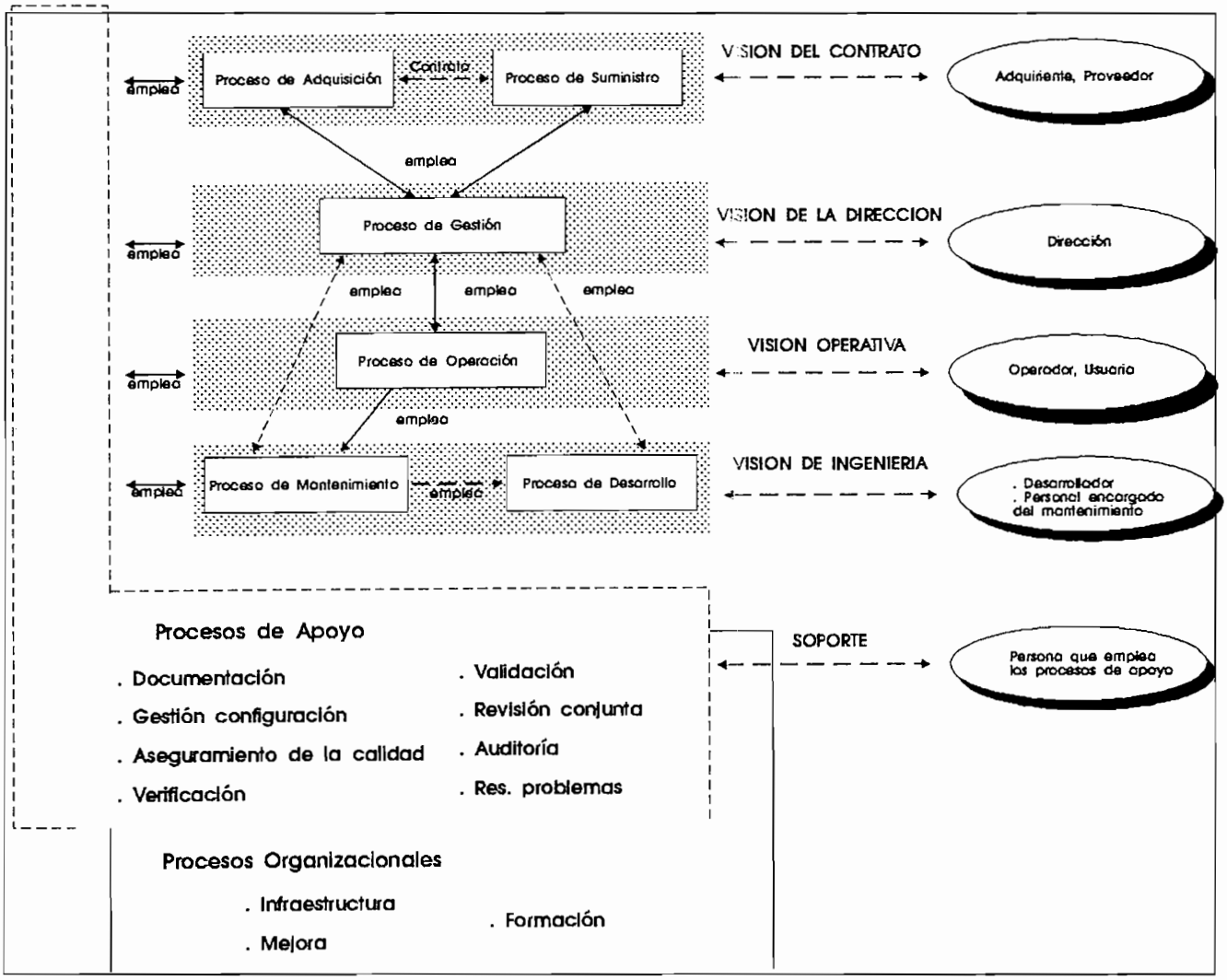


Figura 3: Papeles e interrelaciones de los procesos del ciclo de vida

Ingeniería del Software

Uso de Técnicas de Aprendizaje para la Validación de Especificaciones

Jordi Alvarez, Núria Castell
Departament de Llenguatges i Sistemes Informàtics, Universitat Politécnica de Catalunya

Pau Gargallo 5, 08028 Barcelona
jalvarez, castell@lsi.upc.es

Resumen: *en este artículo abordamos el proceso de validacióndе especificaciones preliminares escritas en lenguaje natural desde una nueva perspectiva: el uso de técnicasde aprendizaje automático. Describimos un sistema que a partir de ejemplos de especificaciones correctas construye o refina un modelo que nos permitirá validar nuevas especificaciones.*

1. Introducción

La fase de ingeniería de requerimientos es una de las más importantes dentro del proceso de desarrollo del software. Se ha comprobado en diversos estudios que la mayoría de los errores detectados en la fase de codificación y mantenimiento provienen de una mala especificación {Basili:84,Endres:75}. Además, el coste de corregir un error de especificación se incrementa desmesuradamente cuando se detecta en las fases de codificación o mantenimiento {Pressman:92}. Esta fue una de las principales motivaciones por las que surgió el proyecto LESD {Castell:94}. Éste define cinco factores que nos permitirían validar la calidad de las especificaciones: trazabilidad, completitud, consistencia, verificabilidad y modificabilidad. LESD es continuado por el proyecto CICYT TIC93-0420 en el cual se enmarca este trabajo.

El artículo intentadar una nueva perspectiva a la validacióndе especificaciones preliminares escritas en lenguaje natural. Tanto el hecho de que se trate de especificaciones preliminares como el que éstas estén escritas en lenguaje natural hace que sea realmente difícil dar una definición clara sobre los criterios que hacen que una especificación sea válida o no. Por ello proponemos usar Técnicas de aprendizaje automático para adquirir, a partir de ejemplos correctos de especificaciones, un modelo de especificación válida.

Para la representación de las especificaciones usamos una**red semántica**. Es un mecanismo que permita representar facilmente estructuras jerárquicas y la mayoría de problemas que se presentan en ingeniería del software son de ese tipo {Maiden:95}; y además las redes semánticas, con la ayuda de los tres**mecanismos básicos de representación (agregación, clasificación y especialización**)de {Borgida:86} constituyen quizás el mecanismo de representación de conocimiento más adecuado para este tipo de problemas {Devanbu:92}.

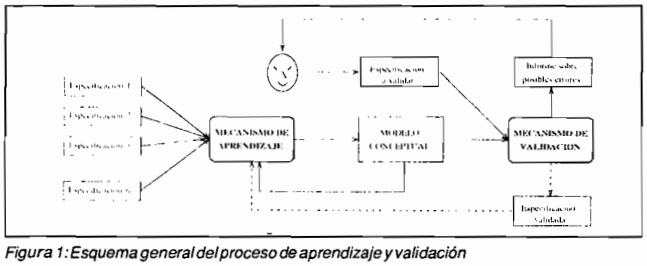


Figura 1: Esquema general del proceso de aprendizaje y validación

De este modo, a partir de especificaciones válidas, que se pueden ver como compuestas por una serie de descripciones de objetos, abstraeremos una serie de conceptos basándonos en esas descripciones. Clasificaremos cada uno de estos objetos como instancia de uno de los conceptos creados, y a partir de ellos aprenderemos una serie de propiedades que podemos interpretar como la descripción genérica de cada concepto. Posteriormente, para validar una especificación nos bastaría con clasificar los objetos cuya descripción constituye la especificación y comprobar si cumplen las propiedades correspondientes al concepto bajo el cual han sido clasificados.

En resumen, lo que pretendemos es generar o enriquecer una base de conocimiento a partir de especificaciones ya validadas; y posteriormente usar esta base de conocimiento para validar (y posiblemente también ayudar en la corrección de) otras especificaciones.

En la**figura 1** podemos ver un esquema general del proceso. El aprendizaje puede ser para crear un modelo a partir de cero o bien para enriquecer un modelo que ya tuvieramos previamente. En cualquier caso se trata de un proceso automatizado en el que el ingeniero de software sólo se encarga de proporcionar los ejemplos. En cambio la validación no es un proceso totalmente automático: tras detectar los posibles errores, se genera un informe que pasa por el ingeniero de software, quien decide si se trata realmente de errores y cómo solucionarlos. Finalmente, una vez que la especificación está validada, puede ser utilizada por el mecanismo de aprendizaje para refinar el modelo conceptual que ya teníamos.

En este artículo nos ocupamos únicamente de los procesos de aprendizaje y validación una vez ya tenemos la representación conceptual de las especificaciones en nuestra red semántica. El paso de las especificaciones tal como las recibimos en lenguaje natural a dicha representación conceptual se describe en{Toussaint:92}; mientras que en {Castell:95} podemos encontrar una descripción global del proceso de control de especificaciones en el cual está inmerso el trabajo descrito aquí. En la sección 2 se comenta la importancia de las estructuras, el elemento básico a partir del cual realizamos el aprendizaje. En la sección 3 distinguimos los diferentes niveles de conocimiento que tenemos en una especificación de requerimientos y sobre los cuales podemos aprender. En la sección 4 profundizamos un poco los mecanismos de

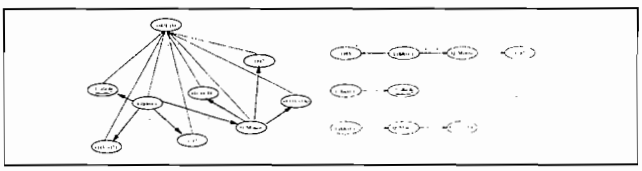


Figura 2: Ejemplo de estructuras básicas a partir de la definición de la representación de un libro y su autor en la red semántica

aprendizaje utilizados, comentando los distintos tipos de concepto que utilizamos para construir una jerarquía y en cómo ésta es generada. En cuanto al aprendizaje en sí mismo, la sección 5 describe el aprendizaje de las propiedades que cumplen las instancias de los conceptos aprendidos; lo que serviría para, tal como se cuenta en la sección 6, validar nuevas especificaciones. La sección 7 describe las condiciones que deben cumplir las especificaciones de ejemplo que proporcionamos al sistema para que el aprendizaje sea el adecuado.

2. Aprendizaje a Partir de Estructuras

Cada **especificación** está formada por una serie de **descripciones de objetos** cada una de las cuales define una cierta interrelación entre ellos. Entendemos como **objeto** cualquier entidad, física o no, que está representada por un nodo en la red semántica. Así p. ej.: actividades, restricciones, condiciones, eventos, etc. serían también objetos.

Cada una de estas descripciones está constituida por una **agregación de descripciones más sencillas** que comunmente se llaman atributos o *slots* y que aquí, para lo que interesa, trataremos siempre como relaciones entre nodos. P.ej. en la **figura 2**, la agregación de las relaciones TITULO, ISBN, AÑO-PUBLICACIÓN Y AUTOR entre LIBRO-1 y los nodos correspondientes de la red semántica constituye la descripción de LIBRO-1. Definiendo una **estructura** como un conjunto de nodos relacionados entre sí, la figura muestra una estructura bastante compleja (8 nodos involucrados sin contar la relación de instanciación) que se puede descomponer entre la descripción de LIBRO-1 y la de Quim Monzó; o también en estructuras lineales más simples (parte derecha de la figura) que son las que utilizaremos en el proceso de aprendizaje.

Esta **agregación**, uno de los tres mecanismos básicos mencionados en la sección anterior, permite construir una representación de la estructura presente en la especificación. Así como {Spanoudakis:94} hace uso de los tres mecanismos mencionados para calcular la similitud entre diferentes estructuras, nosotros sólo usamos directamente el mecanismo de agregación. Aunque el uso que hacemos de él permite, como se verá, sacar provecho de la información que nos puedan proporcionar los mecanismos de clasificación y generalización.

El aprendizaje consistiría en capturar en forma de concepto aquellas estructuras o grupos de estructuras que 'a priori' pensamos que pueden sernos más útiles. De este modo, la extensión de un concepto estaría constituida por aquellos objetos que cumplan determinada propiedad estructural.

Una vez tenemos los conceptos, aprendemos a partir de los ejemplos aquellas propiedades (también estructurales) que cumplen sus instancias (evidentemente serían un subconjunto del total de estas propiedades que cumplen las propiedades que definen el concepto).

Así pues, cada uno de los conceptos llevarían asociadas una serie de propiedades estructurales en las que se distinguen las que definen el concepto y las que están sólo asociadas a él.

Las **propiedades definitorias** establecen una doble implicación entre las instancias de un concepto y éste. Es decir, el hecho de que un objeto esté relacionado de tal forma que cumpla la o las propiedades definitorias de un concepto, es una condición necesaria y a la vez suficiente para que dicho objeto sea instancia de ese concepto. Esto sería muy útil para que posteriormente podamos clasificar automáticamente todos los objetos en la jerarquía aprendida. Las propiedades definitorias están implementadas simplemente como una lista de nodos y relaciones intercalados que el mecanismo de clasificación de la red semántica (no confundir con la 'clasificación concepto-instancia') comprueba cada vez que clasifica un objeto en la jerarquía.

Las **propiedades asociadas** representan propiedades estructurales que las instancias de un concepto acostumbran a cumplir, aunque no son ni una condición necesaria ni suficiente para que un objeto sea instancia de un concepto. Estas propiedades nos servirían para detectar relaciones entre diferentes propiedades estructurales y por lo tanto nos serían útiles para comprobar la completitud y la consistencia de las especificaciones. Las propiedades asociadas están implementadas como relaciones *fuzzy* en la descripción del concepto. Se trata de relaciones entre nodos, como las que aparecen en cualquier red semántica, pero con un peso asociado. A la manera de las conexiones que nos encontramos en las redes neuronales. Esto nos permite guardar el grado en que un concepto cumple una propiedad. Aunque parezca que esto limita las propiedades asociadas que podemos guardar a simples conexiones entre dos nodos, la iteración del proceso de aprendizaje (como veremos más adelante) nos permitiría acceder a propiedades cada vez más complejas.

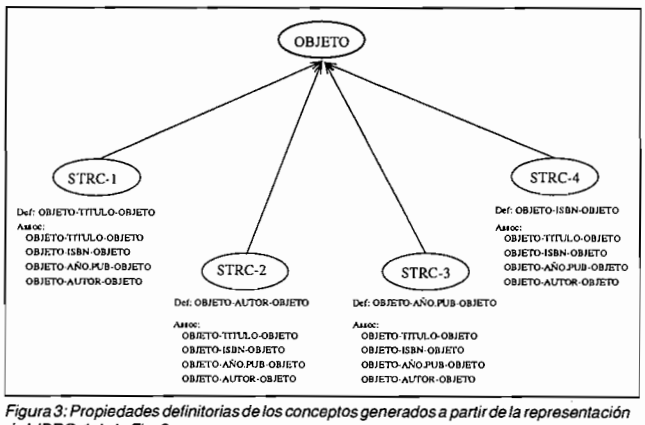


Figura 3: Propiedades definitorias de los conceptos generados a partir de la representación de LIBRO-1 de la Fig. 2

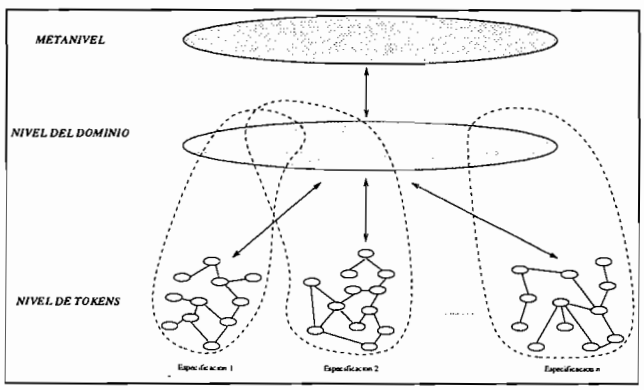


Figura 4: Flujo de información entre los tres niveles de conocimiento

En la **figura 3** podemos observar las propiedades definitorias y asociadas de cuatro conceptos formados a partir de estructuras simples extraidas de la descripción de LIBRO-1.

3. Conocimiento y Metaconocimiento

Hasta ahora hemos hablado de agrupar en conceptos los objetos cuyas descripciones componen las especificaciones; pero no del tipo de conocimiento que el sistema debe aprender. Tipicamente, una especificación contiene dos niveles de información diferentes.

La información sobre objetos tangibles describe la realidad en sí (aquello que tradicionalmente en las redes semánticas se ha dado en llamar token y que en {Dardenne:93} constituye el nivel de instancias). P.ej. en una especificación de un sistema de gestión de una biblioteca, cuando hablamos de 'la biblioteca de la Facultad de Informática de Barcelona', estamos hablando de una entidad presente en la realidad del sistema especificado.

Las descripciones de conceptos del dominio (nivel de dominio en {Dardenne:93}). Siguiendo con el mismo ejemplo, en la especificación de un sistema de gestión de una biblioteca nos encontraremos seguramente con una descripción (adecuada a nuestro problema, eso sí) de librería, libro, etc.

A nivel de representación, el mecanismo de clasificación nos permite distinguir entre estos dos niveles de conocimiento e ir más allá. Si el sistema de representación es lo bastante flexible para tratar los conceptos como simples objetos, puede hacer que cualquier concepto, como objeto que es, sea a la vez instancia de otro concepto. Sistemas como TELOS {Mylopoulos:90} o KAOS {Dardenne:93,Lamsweerde:95} usan esto para colocar un nivel de metaconocimiento por encima del nivel de los conceptos pertenecientes al dominio de aplicación.

La información de este metanivel constituye un metamodelo que permite imponer ciertas restricciones sobre el nivel de conocimiento básico de igual forma que éste hace con el nivel de los *tokens*. Este metanivel constituye pues una herramienta potente para encontrar incompletitudes e inconsistencias en el nivel de descripción del dominio (el modelo).

El aprendizaje nos permite abstraer propiedades que cumplen los objetos de un nivel a partir de objetos del nivel inferior; mientras que el mecanismo de clasificación nos permite tener un modelo de un nivel inferior a partir de su inmediato superior. Debemos ser capaces de aprender tanto sobre los tokens como sobre el modelo. Para ello, seguiremos la misma filosofía que los sistemas de representación mencionados. Tendremos tres niveles: los tokens, el conocimiento del dominio y el metanivel. En {Mylopoulos:90} se menciona la posibilidad de tener más de tres niveles de conocimiento, pero los realmente útiles al menos en este caso son los tres primeros, pues se trata es de modelizar la información que aparece en la especificación y dicha información pertenece a los dos primeros niveles de tokens y de conceptos básicos (la modelización de un nivel viene dada por el nivel inmediatamente superior, el único que puede imponer condiciones sobre él: basta añadir un tercer nivel, que no necesitamos modelizar sino simplemente utilizar).

Así podremos separar el proceso global de aprendizaje en dos niveles distintos: el aprendizaje sobre el **conocimiento del dominio** (a partir del nivel de tokens) y el aprendizaje del **metamodelo** (realizado a partir del conocimiento del dominio). Según nos interese, podemos aplicar el primero, con lo que completaremos y aprenderemos las propiedades de la jerarquía de conceptos del dominio; aplicar el segundo, con lo que generaremos un metamodelo (o completaremos el que ya teníamos); o bien aplicar los dos. De esta forma, tal como se muestra en la **figura 4**, se produce un flujo de información bidireccional (por un lado el aprendizaje y por otro un supuesto sistema de tipos regido por el mecanismo de clasificación) entre el nivel de las instancias y el nivel del dominio {Feather:93}; y entre este último y el metanivel.

4. Generación de Conceptos

La base sobre la cual vamos a construir nuestra jerarquía (bien sea la correspondiente al modelo o la del metamodelo) son las propiedades estructurales de los objetos. Para ello, los conceptos que crearemos se definirían a partir de cadenas estructurales conceptualizadas; entendiendo por cadena estructural una serie de nodos ordenados y unidos cada uno con el siguiente (excepto el último) mediante una relación; y por **conceptualizadas** que aparezca en la cadena, no el objeto que aparece en la especificación inmerso en la estructura, sino el concepto del cual es instancia. En la **figura 5** podemos ver las estructuras simples de la figura 2 conceptualizadas.

El hacer el paso de la conceptualización se debe a que el conocimiento que queremos aprender (recordemos que queremos aprender un modelo para una serie de objetos) está a nivel de conceptos y no a nivel de instancias. Por lo tanto sería una pérdida de tiempo intentar razonar sobre las instancias cuando aportan el mismo conocimiento que los conceptos correspon-

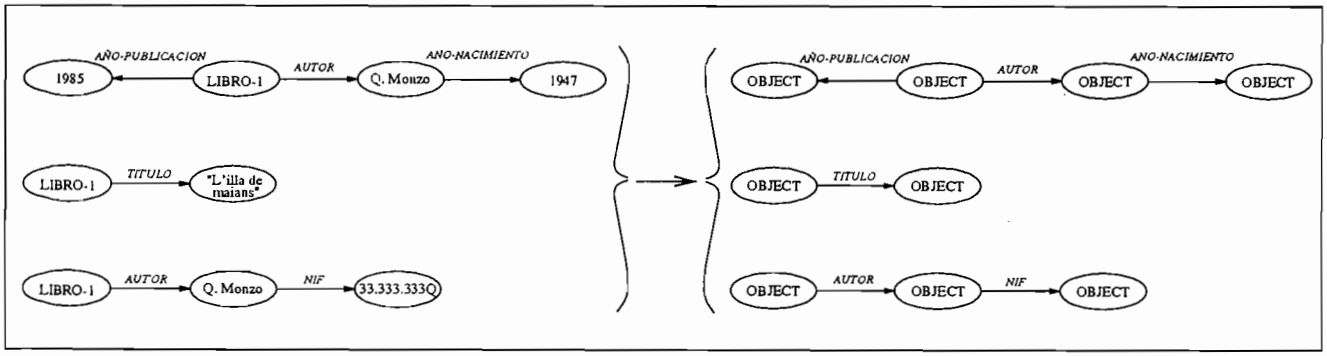


Figura 5: Conceptualización de las estructuras de la fig. 2 teniendo en cuenta que todos los objetos son instancia de objeto

dientes. Son los conceptos quienes dan un significado a sus instancias, que por sí mismas no lo tienen (al principio del proceso de aprendizaje, si partimos de cero, la jerarquía tendría un único concepto al que podemos dar el nombre genérico de object, o meta-object según el caso. Eso significa que al principio medimos todos los objetos por el mismo rasero, y object aparecería como único nodo en todas las posiciones de todas las cadenas estructurales. Esta situación es acorde con que lo único que tenemos para aprender es la estructura si en un principio no tenemos ninguna jerarquía).

La conceptualización de las estructuras es lo que nos permite sacar provecho de los mecanismos de clasificación y generalización. Respecto al primero, su uso en la conceptualización de estructuras es evidente al cambiar los objetos por los conceptos de los que son instancia. En cuanto al segundo, la posterior clasificación en la jerarquía del concepto generado hace que obtengamos estructuras más generales que otras simplemente porque los conceptos y relaciones que participan en unas son más generales que los que participan en las otras. Así por ejemplo, en la **figura 6** podemos ver como los conceptos STRC-1 Y STRC-3 son más generales que STRC-1 Y STRC-3 porque repositorio es más general que biblioteca.

4.1. Tipos de Conceptos

Para construir una jerarquía a partir de cadenas estructurales tenemos tres tipos de conceptos distintos: **conceptos estructurales simples, conceptos conjuntivos y conceptos disyuntivos**. Los primeros vienen definidos por una única propiedad estructural que es una cadena estructural conceptualizada. Los conceptos conjuntivos, como su nombre indica, vienen definidos por la conjunción de las propiedades estructurales que definen a los padres. Y por último, los conceptos disyuntivos se definen a partir de la disyunción de las propiedades estructurales que definen a sus hijos.

Los conceptos estructurales simples constituyen las piezas básicas sobre las que los conceptos conjuntivos y disyuntivos nos permitirían hacer especializaciones y generalizaciones. Con estos tres tipos de conceptos combinados adecuadamente podremos obtener facilmente conceptos con definiciones estructurales complejas. En la **figura 7** podemos ver cómo sirviéndonos de conceptos conjuntivos y estructurales simples construimos una jerarquía en la que especificamos las descripciones mínimas de distintos tipos de publicaciones.

4.2. Analogía con el Aprendizaje Inductivo

Viendo la generación de conceptos desde el punto de vista del aprendizaje inductivo de Michalski y sus reglas de generalización {Michalski:83}, podemos hacer una analogía entre los conceptos generados y las **fórmulas lógicas**. Así, se pueden ver los conceptos estructurales simples como átomos, la conceptualización de estructuras como el paso de constantes a variables, la relación entre los conceptos conjuntivos y sus padres como la aplicación de la regla de eliminación de conjuntandos y la relación entre los conceptos disyuntivos y sus padres como la aplicación de la regla de adición de disyuntandos. Desde este punto de vista, cada uno de los conceptos es en realidad una función de reconocimiento potencial de las propiedades que queremos aprender. No nos debe importar que algunas reglas de generalización de las usadas no preserven la equivalencia lógica entre una fórmula y su generalización (de las tres mencionadas, la única que no la preserva es la de paso de constantes a variables) {Kodratoff:88}. El aprendizaje real de las propiedades se realiza sobre los conceptos generados en una fase posterior; y puede revelarnos algunos de estos conceptos como totalmente inútiles. Lo que sí nos debe preocupar es que hayamos generado como mínimo todos aquellos conceptos que representen verdaderamente funciones de reconocimiento sobre las propiedades que queremos aprender.

Como veremos, el mecanismo de aprendizaje de las propiedades requiere que el nodo relativo al concepto se encuentre explicitamente representado en la red semántica. De no ser así, el aprendizaje se realizaría sobre conceptos representativos de funciones de reconocimiento que por muy parecidas que sean a la verdadera, caerían en la incompletitud o inconsistencia.

La única manera que tenemos de asegurar que todos los conceptos que representen funciones de reconocimiento que nos interesan sean generados es creando todos los conceptos posibles. Pero esto es computacionalmente imposible dado el elevado fórmula de estructuras distintas con que nos podemos encontrar. Por ello, no nos queda más remedio que utilizar heurísticas que nos guien en la generación de conceptos.

4.3. Modelo Heurístico de Generación de Conceptos

Este es un punto que todavía tenemos que explorar en profundidad; si bien hasta este momento hemos utilizado un sistema bastante sencillo que funciona aceptablemente para conocimiento generado aleatoriamente a partir de un metamodelo previo que es destruido justo antes de empezar el aprendizaje (parecido al utilizado en KAOS {Dardenne:93}).

El proceso consiste en generar tantas estructuras simples como aparezcan en las descripciones de ejemplos que tengamos. Para no disparar la complejidad de los cálculos las estructuras generadas no deben sobrepasar una longitud máxima preestablecida. De todas maneras, si con estructuras de una determinada longitud no tuviéramos suficiente, la

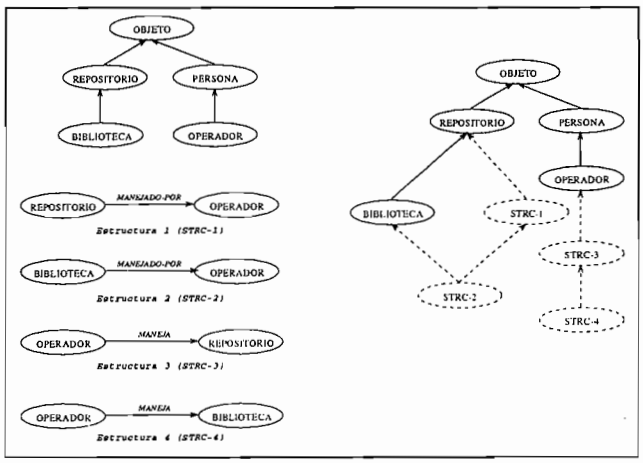


Figura 6: Clasificación de estructuras según los conceptos que participan en ellas. A la izquierda, una simplificación de la jerarquía inicial y las estructuras que vamos a introducir. A la derecha podemos ver como queda la jerarquía después de introducir los nuevos conceptos y clasificarlos automáticamente.

iteración del proceso nos permitiría, tal como veremos más adelante, acceder a estructuras más largas de forma automática.

Tras obtener las estructuras simples, usamos conceptos conjuntivos para generar una estructura compleja para cada descripción de objeto que tengamos y luego clasificamos cada objeto como instancia del concepto correspondiente. La herencia múltiple puede realizarse siempre a nivel conceptual; lo que permite al mecanismo de aprendizaje abstraer propiedades relacionadas con la conjunción de varios conceptos simples.

Para terminar de construir la jerarquía usamos ***técnicas de clustering*** {Jain:88} sobre las estructuras básicas generadas. La distancia de un concepto a un *cluster* se calcula como la máxima de sus distancias a todos los elementos del cluster. La distancia entre distintos conceptos se calcula a partir de la descripción de cada concepto, que viene dada por el conjunto de propiedades asociadas a él. Por lo tanto, es necesario haber realizado el aprendizaje de propiedades, explicado en la próxima sección, sobre los conceptos a los que se aplica el clustering.

De forma intuitiva, la distancia (y también la similitud) entre las diferentes descripciones de conceptos se calcula a partir de la información común (propiedades asociadas iguales) que tienen, penalizando la ausencia de información. Esta ausencia de información se mide a partir del total de tipos de propiedades asociadas que puede tener un concepto. Su penalización hace que la supuesta distancia entre dos elementos no sea tal distancia sino simplemente un heurístico; ya que una de las propiedades que debe cumplir es que la distancia entre un elemento y él mismo sea siempre nula {Kodratoff:88}. En nuestro caso sólo sería nula si su descripción contiene todos los tipos de propiedades asociadas existentes. De todas maneras, esta penalización proporciona una medida heurística que se muestra como la mejor de las que hemos probado. Además, fuerza a construir una jerarquía de forma que los objetos con mucha información (ya sean conceptos o instancias de ellos) estarían situados en la parte profunda de la jerarquía, mientras que los que tienen poca información estarían situados más arriba.

No generamos un único nivel de clusters sino que empezamos con un umbral bajo de distancia (podemos empezar con cero, por ejemplo) y vamos iterando el proceso mientras incrementamos este umbral hasta llegar a 1 (la máxima distancia entre dos conceptos). De esta forma, acabaremos generando un único cluster en la parte más alta de la jerarquía que se correspondería con la raíz de ésta, con lo cual la habremos completado.

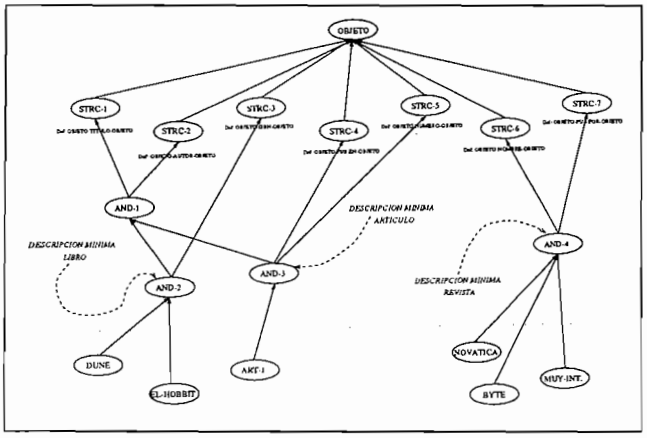


Figura 7: Distinción de distintos tipos de publicación mediante el uso de propiedades estructurales definitorias

5. Aprendizaje de las Propiedades

Una vez que ya tenemos los nodos explícitamente representados en la red semántica, procedemos con el aprendizaje de las propiedades estructurales asociadas a cada concepto. Esta fase se puede realizar bien cuando los conceptos están ya todos generados, o bien incrementalmente a medida que se van generando (de hecho, las técnicasde clustering del apartado anterior requieren que en los conceptos sobre los que se aplica el *clustering* ya se haya hecho la abstracción de las propiedades asociadas).

Como se puede observar en la **figura 8**, las propiedades, tanto definitorias como asociadas y por complejas que sean, se pueden reducir a propiedades asociadas más simples entre conceptos más complejos. Por ello podemos reducir la representación de tales propiedades a la representación de simples *links* o relaciones (con un peso que indica el grado en que la propiedad se cumple). Y por tanto, podemos reducir también el aprendizaje al aprendizaje de simples *links*. En la figura se puede observar como tras la primera iteración del proceso (dibujada con trazo de linea continua) se genera la propiedad definitoria OBJETO-AUTOR-OBJETO para el concepto STRC-1. En la segunda iteración (trazo punteado) generamos un nuevo concepto que especializa la propiedad definitoria de STRC-1 A STRC-1-AUTOR-AND-1. Y como AND-1 es un concepto definido a partir de propiedades estructurales, la propiedad definitoria de STRC-5 equivale a la conjunción de: STRC-1-AUTOR-OBJETO-NIF-OBJETO, STRC-1-AUTOR-OBJETO-TELEFONO-OBJETO y STRC-1-AUTOR-OBJETO-NOMBRE-OBJETO. Lo mismo ocurre para las propiedades asociadas ya que aparecen en ellas cada vez nodos con definiciones estructurales más complejas a base de iterar el proceso de aprendizaje.

En definitiva, el aprendizaje consiste pues en decidir en qué grado los conceptos generados están en relación con otros conceptos. Para ello, inspirándonos libremente en el modelo propuesto por Shastri en {Shastri:88}, damos a cada *link* entre dos conceptos el valor correspondiente al porcentaje de instancias del primer concepto que están relacionadas mediante

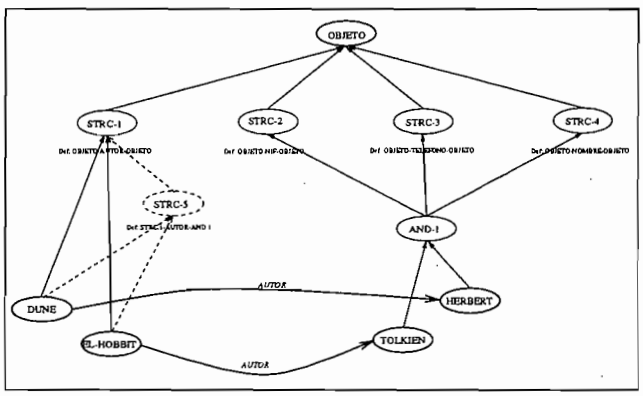


Figura 8: Iteración del proceso de generación de conceptos

este tipo de *link* a instancias del segundo concepto. De esta forma, suponiendo que los nodos de nuestra red semántica se activaran y que cuando se activara un objeto se activaran todos los conceptos del cual es instancia, el peso de un *link* es una medida de la frecuencia de activación del nodo destino se activa cuando el nodo origen está activado; lo cual se ajusta extremadamente bien a una interpretación Hebbiana de los pesos sinápticos en las redes neuronales {Shastri:88}. Pero a diferencia de Shastri, nuestros *links* están etiquetados (sus etiquetas son las relaciones); por lo que podemos tener varios *links* diferentes entre los mismos dos conceptos.

6. Validación de Especificaciones

Una vez tenemos realizado el proceso de aprendizaje, podemos empezar a validar especificaciones. La validación es sencilla. Si interpretamos la especificación que vamos a validar como un conjunto de descripciones de los objetos que intervienen en el sistema especificado, sólo necesitamos clasificar en la jerarquía aprendida cada uno de estos objetos y posteriormente comparar sus propiedades con las del concepto bajo el cual ha sido clasificado.

Las propiedades definitorias de los conceptos servirían para clasificar los objetos, mientras que las propiedades asociadas servirían para verificar que su descripción sea completa y consistente.

Si un concepto tiene alguna propiedad estructural asociada que alguna de sus instancias no tiene, habremos detectado una posible incompletitud (mas adelante ahondaremos en el porqué del término posible). La razón es sencilla: si un grupo de objetos que tienen en común parte de su estructura (las propiedades estructurales definitorias del concepto que los agrupa) tienen todos ellos (o la mayoría) cierta propiedad y nos encontramos con otro objeto que tiene en común toda la estructura menos esta última propiedad, sospecharemos la posibilidad de que la descripción de este objeto sea incompleta. La evidencia que tengamos de esto dependería del porcentaje de instancias que cumplan la propiedad. Reciprocamente, si un objeto tiene alguna propiedad que el concepto bajo el cual ha sido clasificado no tiene, sospecharemos la existencia de una posible inconsistencia o bien que la descripción del objeto contiene información redundante.

Pero no todo termina ahí. Hay otro caso que permite detectar una posible incompletitud. ¿Qué pasa si la descripción de un objeto es tan pobre (y tan incompleta) que hace que la clasificación se quede a medias? Al estar clasificado bajo un antecesor del concepto bajo el cual tendría que estar clasificado, es muy posible que no detectemos buena parte de las incompletitudes.

Una solución directa a este problema es hacer un proceso iterativo: cada vez que vamos completando la descripción del objeto y eliminando incompletitudes, reclasificamos el objeto, con lo que probablemente encontremos nuevas incompletitudes. Así hasta que ya no encontramos más. Este proceso nos serviría para la mayor parte de los casos, pero no podemos asegurar que nos sirva siempre. Para asegurarnos, existe una solución complementaria. En el caso de que nuestro objeto haya sido clasificado bajo una concepto que no tiene instancias directas, sino que sus instancias son instancias de algún concepto más específico (lo que en terminología de lenguajes orientados a objetos se llamaría una clase abstracta), podemos deducir que su descripción es tan incompleta que no permite una clasificación total del objeto. De esta forma, si el -llamémosle- concepto abstracto no permite iniciar el proceso iterativo que acabamos de comentar, podemos intentar guiarnos por las propiedades definitorias de los conceptos hijos. Es decir; intuimos que el objeto no puede ser instancia directa del concepto supuestamente abstracto, por lo que debe ser instancia (directa o no, da igual) de alguno de sus hijos. Por lo tanto, intentamos averiguar qué propiedad o propiedades definitorias de los hijos debería cumplir el objeto y evidentemente no cumple (si no, la clasificación hubiera continuado).

En la **figura 9** podemos ver como una descripción incompleta de un libro se clasifica bajo AND-1. Al no tener éste ninguna instancia directa, el sistema sospecharía que la descripción es incompleta y propondría al usuario completar la descripción de EL-HOBBIT con propiedades correspondientes a AND-2 y AND-3 (según lo que conoce el sistema, la descripción del objeto puede evolucionar hacia un libro o hacia una revista).

Mientras en los otros casos de detección de posibles incompletitudes e inconsistencias, estaba claro cuales eran las propiedades estructurales en cuestión, aquí tan sólo sabemos que se trata de un subconjunto de las propiedades estructurales que diferencian a los hijos del concepto abstracto de éste. Es el ingeniero de software quien debe decidir cuales son las propiedades que faltan (si es que realmente falta alguna).

7. Conjunto de Entrenamiento

En cualquier método de aprendizaje a partir de ejemplos, tiene vital importancia la elección del conjunto de ejemplos a partir de los cuales vamos a realizar el entrenamiento (conjunto de entrenamiento). En nuestro caso, el conjunto de entrenamiento está formado por especificaciones. Después de hablar de la validación en el apartado anterior, analizaremos qué propiedades deben cumplir las especificaciones del conjunto de entrenamiento para que el aprendizaje y la posterior validación sean eficaces. Y para que el aprendizaje sea eficaz, todas en conjunto y cada una en particular deberán cumplir algunas propiedades que comentamos a continuación.

7.1. Representatividad

En primer lugar, las especificaciones deben ser ampliamente representativas dentro del dominio que estamos tratando. Si todas ellas se restringen a un subconjunto del dominio, tendremos una parte del dominio sobre la que no podremos aprender nada. Esto puede dar lugar a dos errores distintos. Por un lado, cuando el sistema se encuentra con una estructura (o conjunción de estructuras) que no ha encontrado anteriormente, dará un aviso de inconsistencia. Por otro lado, cuando el sistema encuentra que falta una estructura que siempre ha visto, dará un aviso de incompletitud.

Evidentemente no podemos pretender que cualquier situación posible en nuestro dominio esté presente en alguna de las

especificaciones de ejemplo, con lo que no podemos asegurar que las incompletitudes e inconsistencias detectadas por el sistema lo sean de verdad. Por ello, la decisión final la debe tomar siempre el ingeniero de software {Reubenstein:91} y debemos entender los avisos del sistema de validación como lo que son: avisos sobre posibles incompletitudes o inconsistencias que el sistema deja decidir si realmente lo son o no al buen juicio del ingeniero de software .

Todo esto no quiere decir que el sistema presentado no sirva para nada. Después de un extenso análisis de problemas propios de ingeniería del software, {Maiden:95} argumenta que la mayor parte de ellos pertenecen a un conjunto tratable de clases jerárquicas. Si las abstracciones realizadas en la fase de aprendizaje son suficientemente buenas, y para un mismo dominio de aplicación y entorno de desarrollo, el número de situaciones nuevas que puede presentar una especificación respecto a un conjunto de ellas (el de entrenamiento) decrece rapidamente si el tamaño del conjunto crece. Además, una vez validadas, las nuevas especificaciones se pueden ir incorporando al conjunto de entrenamiento, con lo que los posibles avisos erróneos se van corrigiendo y la fiabilidad del sistema de validación va en aumento.

7.2. Consistencia

Evidentemente, cada una de las especificaciones debe ser consistente. De no ser así, el sistema aprenderá las inconsistencias de los ejemplos y el resultado de la validación perderá cualquier valor que pudiera tener.

7.3. Minimalidad

Importa que las especificaciones que formen parte del conjunto de entrenamiento sean mínimas. Así nos aseguramos que todas las propiedades que aprende el sistema son estrictamente necesarias en la especificación. Esto es especialmente importante para comprobación de la completitud, ya que justifica que el sistema pueda asegurar que existe una incompletitud cuando detecta la ausencia de una de esas propiedades. De no ser mínimas las especificaciones, podríamos aprender propiedades innecesarias, lo que haría que el sistema pudiera detectar incompletitudes allí donde no las hay.

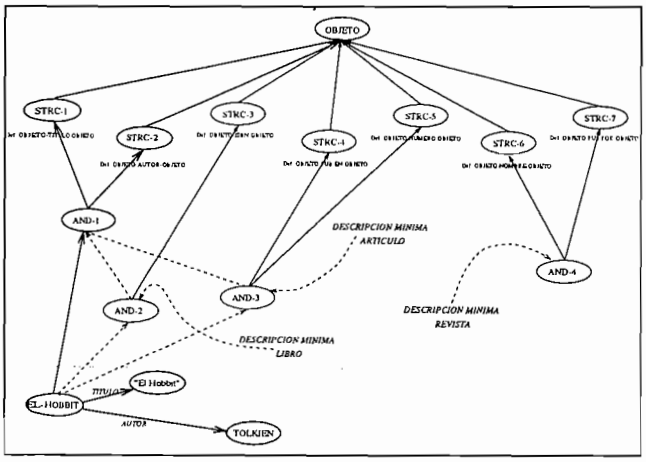


Figura 9: Validación de una descripción de libro a la cual le falta el atributo ISBN.

7.4. Completitud

Reciprocamente, también es importante que las especificaciones sean completas. Si no, el sistema podría aprender descripciones incompletas de objetos y por tanto detectar menos incompletitudes de las realmente existentes. Además, la incompletitud de las especificaciones del conjunto de entrenamiento también afectaría a la detección de inconsistencias. El sistema no aprendería algunas de las propiedades necesarias, y cuando se las encontrara, las reportaría como inconsistencias.

7.5. Relajación de las Condiciones

Como para realizar el aprendizaje de las propiedades asociadas a cada concepto no se utilizan técnicas de aprendizaje simbólico sino que únicamente se mantiene un ratio del porcentaje de instancias que cumplen dichas propiedades, el sistema es bastante robusto y resistente a errores. Por ello, no es necesario que las especificaciones del conjunto de entrenamiento cumplan al cien por cien las condiciones especificadas. El sistema construirá un modelo de validación más o menos correcto aunque las especificaciones no sean totalmente correctas. Eso sí, cuantos más errores, más ejemplos necesitaremos para obtener una validación fiable.

8. Conclusiones y trabajo Futuro

Hemos presentado en este artículo un sistema que a partir de especificaciones ejemplares es capaz de abstraer un modelo de especificación para comprobar la completitud y la consistencia de nuevas especificaciones.

La construcción manual de una base de conocimiento que permita validar especificaciones resulta altamente inviable y costosa debido a la complejidad del problema (sobre todo en lo que se refiere a la completitud), a la total vinculación de los términos completitud y consistencia al dominio de aplicación y a la metodología usada para el desarrollo del software en cada caso concreto, así como al elevado grado de informalidad que implica el que se trate de especificaciones preliminares escritas en lenguaje natural. Por ello, el uso de Técnicas de aprendizaje automático para la construcción de dicha base de conocimiento parece una buena solución.

Por el momento hemos realizado pruebas satisfactorias, generando automáticamente conocimiento a partir de un metamodelo parecido al usado en el sistema KAOS y aprendiendo a partir del conocimiento generado. El resultado es que se aprenden (entre otros) los mismos conceptos que estaban en el metamodelo y con las mismas propiedades.

En un futuro próximo planeamos usar especificaciones reales, lo cual nos serviría para comprobar si el sistema es capaz de abstraer un modelo y un metamodelo a partir de ejemplos sobre los que no tenemos un metamodelo previo. Esto nos ayudará a hacer un estudio más profundo sobre el conjunto de heurísticas que utilizamos para generar la jerarquía. Puede resultar interesante añadir algunas reglas más de generalización de Michalski, como por ejemplo la de subir en el árbol de generalización, lo que nos permitiría realizar automáticamente la generalización de la figura 6.

Por otro lado, para incrementar la potencia del sistema de representación, puede ser interesante introducir en la propia estructura conceptos reflexivos sobre ésta misma. Por ejemplo, podríamos introducir el concepto SELF para referirse de una manera estándar al primer elemento de la estructura. Esto permitiría entre otras cosas descubrir propiedades de las relaciones temporales, comprobando que la cadena estructural CONCEPTO—ANTES—CONCEPTO—ANTES—SELF nunca se cumple y por tanto es inconsistente que A pase antes que B si B pasa antes que A. A un nivel más abstracto podríamos aprender qué relaciones son inversas de otras mediante la cadena: CONCEPTO—RELACION—CONCEPTO—INVERSA—SELF.

Por último, también pensamos trabajar sobre un sistema que permita ir eliminando todo el conocimiento generado que no nos sea útil. El sistema genera gran cantidad de conceptos y conexiones que después resultan inútiles. Si para problemas reales no se elimina buena parte de esta 'basura', la cantidad de espacio ocupado puede ser desmesurada.

Bibliografía

V.Basili, B.Perricone. *Software errors and complexity: An empyrical investigation.* Communications of the ACM, 27(1):42—52, Jan. 1984.

A.Borgida. *Conceptual modeling of information systems.* In M.Brodie and J.Mylopoulos, editors, Knowledge Base Management Systems, chapter 31, pages 461—469. Springer-Verlag, 1986.

N.Castell, A.Hernández. *Filtering software specifications written in natural language.* In Proceedings of 7th Portuguese Conference on Artificial Intelligence, (EPIA '95), pages 447—455, Madeira, Portugal, Oct. 1995. LNAI-990, Springer Verlag.

N.Castell, O.Slavkova, Y.Toussaint, A.Tuells. *Quality control of software specifications written in natural language.* In Proceedings of 7th International Conference on Industrial & Engineering Applications of AI & Expert Systems (IEA-AIE'94), pages 37—44, Austin, Texas, June 1994. Gordon and Breach Science Publishers.

A.Dardenne, A.van Lamsweerde, S.Fickas. *Goal-directed requirements acquisition.* Science of Computer Programming, 20:3—50, 1993.

P.Devanbu, R.Brachman, P.Selfridge, B.Ballard. *Lassie: A knowledge-based software information system.* Communications of the ACM, 34(5):35—49, May 1991.

A.Endres. *An analysis of errors and their causes in system programs.* IEEE Transactions on Software Engineering, 1(2):140—149, June 1975.

M.Feather. *Requirements reconnoitering at the juncture of domain and instance.* In Proceedings of 1st International Symposium on Requirements Engineering, pages 73—76, San Diego, California, Jan. 1993. IEEE Computer Society Press.

A.Jain, R.Dubes. *Algorithms for Clustering Data.* Prentice-Hall, 1988.

Y.Kodratoff. *Introduction to Machine Learning.* Morgan Kaufman Publishers, 1988.

N.Maiden, P.Mistry, A.Sutcliffe. *How people categorise requirements for reuse: a natural approach.* In Proceedings of 2nd International Symposium on Requirements Engineering, pages 194—203, York, England, Mar. 1995. IEEE Computer Society Press.

R.Michalski. *A theory and methodology of inductive learning.* In R.Michalski, J.Carbonell and T.Mitchell, editors, Machine Learning, An Artificial Intelligence Approach, pages 83—130. Morgan Kaufman Publishers, 1983.

J.Mylopoulos, M.Jarke, A.Borgida, M.Koubarakis. *Telos: Representing knowledge about information systems.* ACM Transactions on Information Systems}, 8(4):325—362, Oct.1990.

R.Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New York, NY, USA, third edition, 1992.

H.Reubenstein, R.Waters. *The requirements apprentice: Automated assistance for requirements acquisition.* IEEE Transactions on Software Engineering, 17(3):226—240, Mar.1991.

L.Shastri. *A connectionist approach to knowledge representation and limited inference.* Cognitive Science}, 12:331—392, 1988.

G.Spanoudakis, P.Constantopoulos. *Measuring similarity between software artifacts.* In Proceedings of 6th International Conference on Software Engineering and Knowledge Engineering, Jurmala, Latvia, June 1994.

Y.Toussaint. *Méthodes Informatiques et Linguistiques pour l'aide a la Spécification de Logiciel.* PhD thesis, Paul Sabatier University, Toulouse, 1992.

A.van Lamsweerde, R.Darimon, P.Massonet. *Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt.* In Proceedings of 2nd International Symposium on Requirements Engineering, pages 194—203, York, England, Mar. 1995. IEEE Computer Society Press.

Agradecimientos

Este trabajo está parcialmente subvencionado por la CICYT (TIC93-420) y por la CIRIT (GRQ93-3015 y una beca predoctoral a Jordi Alvarez).

Ingeniería del Software

Ignacio García Benito
Servicio Informática CEH, Junta de Andalucía
José Antonio Mellado Jiménez
SADIEL, S.A.

Calidad en el mantenimiento del sistema de gestión económica de la Junta de Andalucía

1. Introducción

El Sistema Júpiter es el sistema corporativo de la gestión económica de la Junta de Andalucía. El núcleo central del sistema fue implantado en enero de 1.993 y desde entonces se han ido incorporando nuevos subsistemas que han abarcado diversas áreas: Tesoría, Contable, Presupuestaria y Planificación. Las dimensiones del sistema pueden apreciarse por las siguientes cifras:

- Número de programas a mantener : 4.000 (on-line + batch)
- Número de miembros del equipo de mantenimiento : 27
- Usuarios potenciales : 1.200
- Usuarios concurrentes : entre 200 y 300.
- Extensión territorial : Comunidad Autónoma de Andalucía.
- Nº de demandas registradas/año:

Relacionadas con datos:	665
Errores:	555
Mejoras:	1.244
Nuevas:	4.208

2. Establecimiento de un Equipo de Garantía de Calidad

Siguiendo las indicaciones de METRICA versión 2, se formó un equipo de Garantía de Calidad con las siguientes funciones:

- Aseguramiento para la dirección del Servicio de Informática sobre mantenimiento de la uniformidad en los procedimientos de Análisis y Diseño de las nuevas funciones requeridas.
- Control de versiones y del paso al entorno de Explotación de los componentes modificados.
- Implantación de métricas para conocer con mayor exactitud el grado de avance de los trabajos.
- Propuesta de normas de trabajo.
- Supervisión, seguimiento, control y actualización de las normas.
- Mantenimiento y promoción de una biblioteca de módulos reutilizables.
- Desarrollo de un sistema propio de control de tareas.

En los dos años que lleva funcionando este equipo se ha avanzado en la implantación de Calidad en distintos frentes:

- Control automático de tareas
- Rediseño e incremento de modularización
- Actualización de la documentación
- Control de calidad previo al paso a explotación

3. Gestión: Control Automático de Tareas

En una primera fase se implantó un sistema informático que tenía como usuario final el propio personal informático y cuya finalidad es establecer un control de todos los trabajos que se realizan: desde la detección de la necesidad hasta la conformidad del usuario. El sistema fue desarrollado en NATURAL + ADABAS, el entorno de programación del Sistema Júpiter.

Cada nueva tarea se introduce en el ordenador por el responsable del área junto (con un texto explicativo de la modificación a realizar) y se le asigna un código de forma automática. La tarea puede ser simple o estar dividida en subtareas, cada una de las cuales es asignada directamente a un analista-programador o bien, en caso de mayor dificultad, a un analista para que, tras realizar el análisis de las repercusiones de los nuevos requisitos, la distribuya a los programadores que se encargarán de desarrollar el código. Tanto el analista como el programador o analista-programador deben introducir cada día el tiempo dedicado a esa tarea hasta su finalización. Este sistema ha permitido disponer de estadísticas para la comparación de productividades, distribución de recursos por área, nº tareas en curso, lista de tareas pendientes, etc.

Una consigna ha facilitado la aplicación rigurosa del sistema de tareas : "No se hace ningún trabajo sin que se disponga del

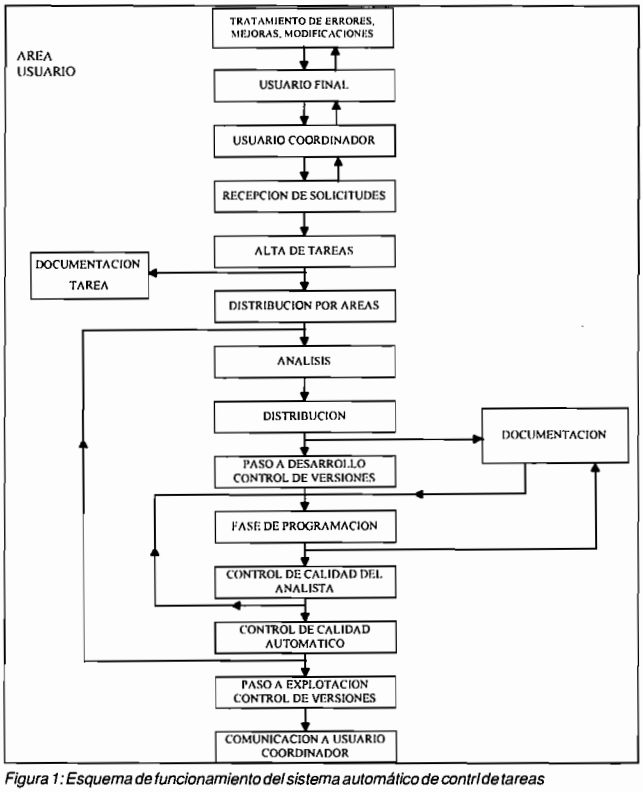


Figura 1: Esquema de funcionamiento del sistema automático de contrl de tareas

código y asignación de la tarea a la que se ha de imputar el tiempo que se le dedique". Esto, lejos de suponer una paralización del funcionamiento del equipo de trabajo, ha supuesto una ordenación de las tareas a realizar y una clarificación de los papeles de cada componente del equipo. Cada mañana, al conectarse al ordenador, el programador conoce la lista de tareas pendientes, y el sistema le 'invita' a que se de por enterado de las nuevas tareas que debe realizar. El éxito ha sido tal que, una tarea 'nula' que se había previsto para asignar los tiempos muertos, por descoordinación o por falta de asignación de tarea, ha tenido un uso absolutamente marginal.

El circuito queda completo con la elaboración de la documentación por parte de quien realiza la tarea, detallando la solución al problema planteado y su devolución al responsable que la distribuyó. Éste debe supervisar la bondad de la solución realizada y comprobar el correcto funcionamiento del código modificado.

4. Rediseño: Incremento de la modularización del software e implantación de una biblioteca de módulos comunes

Existían dos fuentes continuas de problemas a la hora de realizar los cambios en el software : el excesivo 'personalismo' del código y la repetición no bien controlada del mismo.

Por excesivo personalismo de un código se entiende el que, con su sola lectura, permite determinar la persona que lo ha escrito, esto es, el polo opuesto a la pretendida uniformidad en los desarrollos. Con una estructura formal predefinida de los programas del mismo tipo, se consigue que cualquier programador pueda leer con facilidad el programa escrito por otro y realizar modificaciones con mayor agilidad, pues ambos escriben programas en el mismo estilo. Esta afirmación, por muy repetida que se encuentre, no puede dejar de tenerse de nuevo presente cuando de lo que se trata es de prestar un servicio de mantenimiento de software.

Se ha realizado una labor de formación con el fin de promover la programación modular, que, al mismo tiempo que facilita la comprensión de cómo realiza su función un determinado programa, permite ir formando una biblioteca de módulos reutilizables que, sin duda, hará más rápidos y fiables los desarrollos futuros. Hay que tener en cuenta que la llamada a un módulo que ofrece garantías de funcionamiento (probado exhaustivamente), ahorra un tiempo considerable en las pruebas del nuevo programa, al no ser necesario realizar comprobaciones sobre las funciones propias del módulo llamado: sólo es necesario comprobar que la interfaz funciona de forma correcta.

Cada módulo debe resolver un problema concreto de diseño y se ha de conseguir que las soluciones a los problemas planteados sean lo más sencillas posible, evitando soluciones ingeniosas que, aunque ahorran código, repercuten de forma muy desfavorable en la claridad del programa.

La reutilización de módulos, al evitar la proliferación de soluciones alternativas a un único problema y concentrar en un mismo código la solución, facilita en un alto grado el mantenimiento del sistema. Los módulos disponibles se encuentran actualmente en una librería separada, donde se han clasificado por temas y se han establecido palabras clave para su localización.

Ahora bien, los criterios de modularización no eran fijos. Se detectaban también diferentes 'estilos' a la hora de establecer particiones que pudieran ser reutilizables.

La uniformidad en la modularización se ha logrado establecer gracias a la aplicación de algunas ideas basadas en conceptos de la Orientación a Objetos. Se han seguido los siguientes pasos:
- Establecimiento de clases y subclases.
- Identificación de operaciones en cada clase
- Establecimiento de escenarios para comprobar la viabilidad de montaje de las distintas funciones a partir de mensajes entre clases que activen las operaciones necesarias.

En la **figura 2** aparece el escenario correspondiente a la consulta de una factura.

Al no disponer de un lenguaje de programación orientado a objetos, este enfoque ha llevado a la creación de numerosos subprogramas, especializados en su función, que se comunican por medio del área de parámetros.

5. Actualización de la Documentación

El acceso a la documentación del sistema se ha simplificado de forma significativa desde que el equipo de trabajo dispone de red local, pues toda la documentación se ha centralizado en el servidor. Se ha abierto un directorio por cada subsistema y se han establecido plantillas para la documentación de clases y operaciones.

Las claves de la documentación del sistema son las siguientes:
- Establecimiento de estándares muy detallados de nomenclatura de operaciones: de validación, de existencia, de acceso a la base de datos, etc.
- Establecimiento de un diccionario de palabras clave de búsqueda y utilización del sumario del documento para la localización de operaciones ya realizadas.

Cada operación pasa por los siguientes estados:
- En el momento en que el analista necesita hacer uso de una operación, da por hecho, en principio, que ya existe y la busca en el directorio correspondiente. Si no existiera, la crea con extensión **.opa** (operación pendiente de analizar).
- Una vez analizada, la operación cambia su extensión por **.opp** (operación pendiente de programar).
- Una vez programada la operación queda como **.opc** (operación pendiente de comprobar).
- Por último, una vez que el analista revisa el buen funcionamiento de la operación programada, ésta queda en

estado.**opr** (operación programada y comprobada, lista para ser utilizada).

6. Control de Calidad previo al paso a Explotación

El sistema de control automático de tareas se ha completado con una gestión de componentes. El analista asigna a cada tarea la lista de objetos software que es necesario modificar o crear. El paso a Explotación de los mismos se realiza por cada tarea y no por cada componente, realizándose la comprobación de que la tarea está finalizada y que el programa ha sido debidamente documentado antes de proceder al paso a Explotación.

Con este sistema de tareas y la relación con los componentes implicados se resuelve, además, un problema que hasta ahora había permanecido sin solución: el mantenimiento al día de la documentación. Aún cuando las situaciones de urgencia hayan hecho imposible la actualización del diseño inicial, la consulta de la documentación original más la de las soluciones aportadas a las distintas tareas que han afectado al objeto en cuestión nos garantizan el conocimiento actual del estado del mismo.

7. Conclusión

Los principios de análisis y diseño orientado a objetos se han aplicado con gran ventaja tanto a la definición de nuevos sistemas como al mantenimiento de sistemas ya en Explotación, incorporándoles mejoras en su diseño. Estas ventajas se presentan aunque los sistemas se hayan implementado con lenguajes convencionales, no orientados a objetos (como NATURAL o NATURAL-WINDOWS, en nuestro caso). Por supuesto que en la implantación ha sido necesario renunciar a algunos de los aspectos de la orientación a objetos (fundamentalmente los relacionados con la herencia), pero los resultados obtenidos en cuanto a uniformidad de modularización y a claridad en el diseño han sido muy satisfactorios.

Por otra parte, el sistema implantado de control de tareas ha supuesto una ordenación de los trabajos a realizar y ha permitido el registro automático de todas las actividades del equipo de trabajo. Por esta vía se ha conseguido avanzar en el establecimiento de métricas cuyo seguimiento hace posible una mejor gestión del proceso de desarrollo y, en definitiva, una mejora en la calidad del servicio prestado.

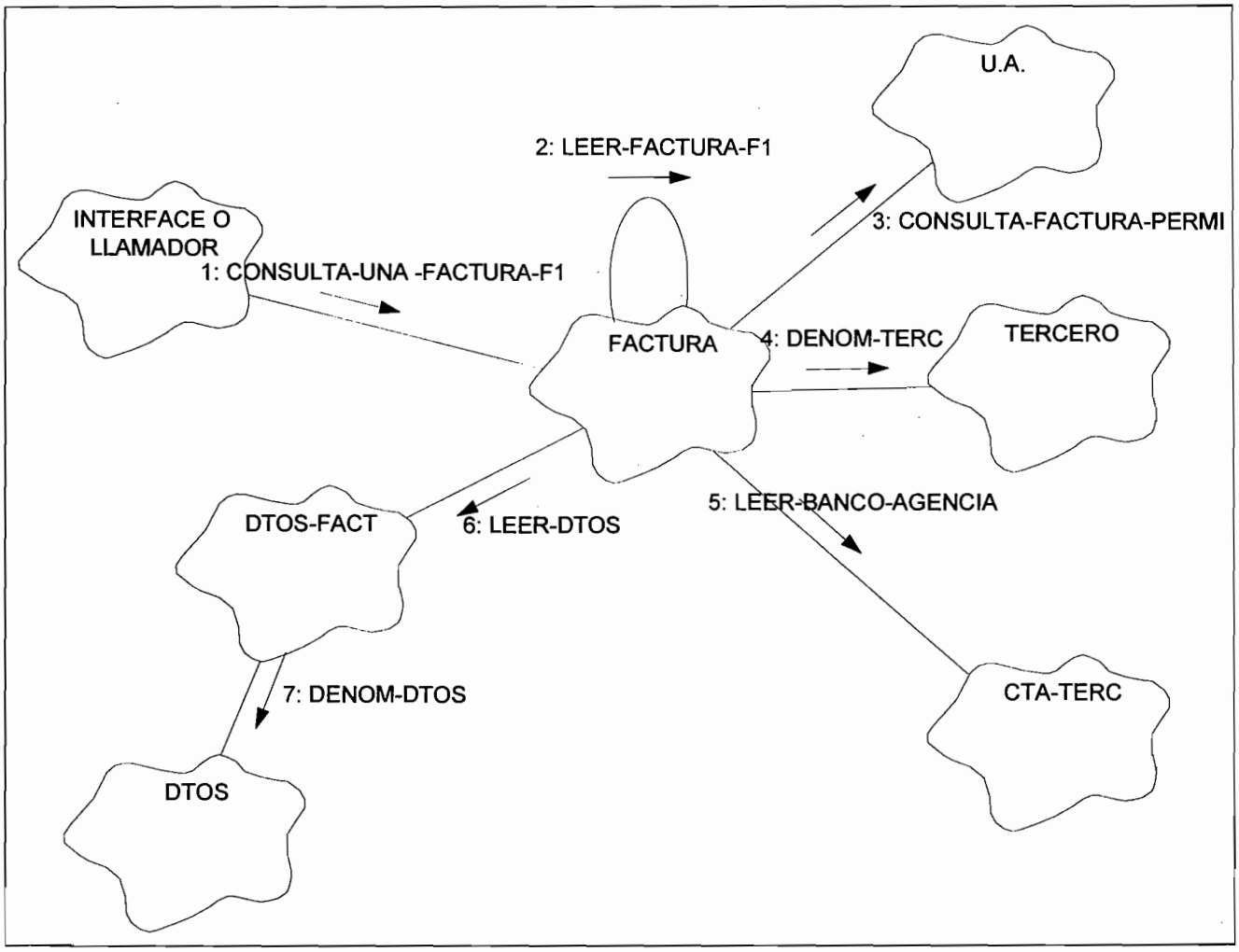


Figura 2: Escenario de la consulta de una factura

Ingeniería del Software

Juan Alberto Balboa Hernández

(Trabajo dirigido por Juan Carlos Granja)

Eurométodo y calidad de software

Resumen: *se plantea la necesidad de establecer un nexo metodológico entre Eurométodo y otras metodologías de desarrollo, incidiendo en un tema tan crucial como es la calidad del software. Para ello intentaremos identificar alguna característica lo suficientemente importante que influya en la calidad de un sistema de información, y que subyazca de forma explícita o implícita en aspectos fundamentales de la filosofía de Eurométodo. Lo que aquí se presenta es un pequeño resumen de la labor realizada por el autor durante un año,* ***dirigida por Juan Carlos Granja*** *y trabajando sobre documentación muy escasa, debido al momento por el que se atravesaba en este tema*

1. Introducción

Dada la situación actual europea respecto a métodos de desarrollo de sistemas enformación, los adquiridores tienen dificultades para comparar ofertas que presenten enfoques distintos; las sociedades de servicios y los profesionales se ven afectados por la fragmentación del mercado europeo, lo cual induce poco a las inversiones y a la cooperación entre ellos; la débil portabilidad de los métodos actuales dificulta tanto la apertura de los mercados públicos a la competencia europea, como a la movilidad de los actores técnicos europeos; la falta de una orientación comunitaria incide tanto en los programas multinacionales (como **Sprit** o **Eureka**) como en la mejora de la competitividad europea.

Un método para establecer un marco metodológico

Dado el carácter de marco metodológico de Eurométodo y de su intento de servir como base para la armonización de metodologías de desarrollo de sistemas de información, aquí se tiene en cuenta este aspecto inexistente en otros métodos.

El diseño de una aproximación para el desarrollo apropiada no solo está guiado por Eurométodo. El montaje organizacional contendrá estándares y reglas que hay que seguir, y los métodos de desarrollo que se aplican en el proceso de producción guiarán las acciones a través de modelos de procesos y perfiles de productos concretos. En este sentido Eurométodo proporciona valiosos conceptos y líneas de guía como suplemento a otros métodos de desarrollo de sistemas, así como a estándares y reglas locales.

Eurométodo ofrece la ayuda necesaria para establecer puentes entre sus propios conceptos, de una parte, y los conceptos de otros métodos de desarrollo, de la otra. Estas líneas de guía pueden usarse en aplicaciones de Eurométodo para adaptaciones concretas.

En un contexto diferente, estas líneas de guía pueden servir como un marco de trabajo para alcanzar algún nivel de armonización conceptual entre diferentes métodos de desarrollo.

2. Planteamiento del problema

Como se ha visto, Eurométodo contempla con especial interés la **comunicación entre sí** de las personas y grupos de una organización, y de estos con los clientes. Se intenta que:

-Los clientes puedan especificar peticiones de ofertas claras y no ambiguas a los proveedores;
-Los proveedores operen mejor conjunta o competitivamente en proyectos nacionales, extranjeros, europeos o internacionales;
-Los clientes puedan comparar estructurada y objetivamente las ofertas recibidas;
-Los participantes en un proyecto puedan seguir ordenadamente, durante todo el ciclo de vida del mismo, el progreso y la calidad.

Y todo ello dentro de un **plan de proyecto adaptado** a cada caso concreto y que dependerá del contexto y de los objetivos deseados.

Ambos temas van a ser fundamentales para la obtención de sistemas con calidad. Por una parte, un buen entendimiento entre las personas involucradas en un proyecto es indispensable para el buen desarrollo y la calidad del sistema final. Por la otra, si conseguimos valorar de forma eficaz la situación de un problema y, en función de esto, establecer un plan de trabajo expreso para cada caso concreto (aunque nos mantegamos siempre dentro de un marco general de trabajo) el desarrollo del sistema va a ser más fácil y más productivo ya que va a estar más de acuerdo a la situación actual y a los objetivos que se pretenden, se van a limitar los valores de la complejidad e incertidumbre del proyecto (y con ello los riesgos asociados), lo cual influirá en la calidad de la gestión, del desarrollo y del producto final.

La **armonización de los métodos** y el facilitar el **entendimiento entre clientes y proveedores** son dos formas -a diferentes niveles- de asegurar la mejora en la comunicación en todo tipo de intervenciones sobre un sistema de información. Una **aproximación situacional** para definir la forma más conveniente de llevar a cabo el proyecto va a influir de forma decisiva en la calidad del sistema. Por último, la garantía de calidad es una actividad esencial. La implantación de un buen **sistema de calidad** ayudará a una mejor gestión del desarrollo del software, y esto también influirá directamente en los aspectos relacionados anteriormente.

Filosofía

Eurométodo
-Buena comunicación
-Plan adaptado a la situación.
-Sistema de Calidad.

Calidad

3. Una posible vía de solución

Según se ha visto en el Planteamiento del Problema sería interesante estudiar: Aproximación situacional, Relación cliente/proveedor, Puentes (productos de calidad). En orden a conseguir estos objetivos contamos, entre otras cosas, con:

-Un conjunto de conceptos y un metalenguaje comunes, definidos en el documento 'Dictionary'.
-La ayuda necesaria para llevar a cabo las transacciones entre clientes y proveedores, en el documento 'Transaction Model'.
-Actividades y productos de garantía de calidad, bien definidos y clasificados en el documento 'Deliverable Model'.
-La posibilidad de establecer puentes con Eurométodo, desde otros métodos. La información necesaria para esto está contenida en el documento 'Method Bridging Guide'.
-La ayuda necesaria para definir o valorar un plan de proyecto, de acuerdo a las circunstancias. La información pertinente está contenida en el documento 'Strategy Model'.

A pesar de que probar su efectividad es difícil y de que la subjetividad no puede eliminarse totalmente en un área de este tipo, algunos trabajos recientes realizados en Escandinavia [Rop], [Saar] y en Holanda [Brink] han iniciado estudios estadísticos sobre la efectividad de diferentes aproximaciones en diferentes contextos.

4. Productos de aseguramiento de la calidad

Como apoyo a los puntos anteriores y teniendo en cuenta que estamos centrados en el tema de la calidad estudiaremos este asunto desde el punto de vista de los productos, que son la base para el establecimiento de puentes y lo que se intercambian clientes y proveedores.

Para Eurométodo la Garantía de Calidad es una de sus tres áreas de trabajo. Ésta consiste en todas aquellas acciones planificadas y sistemáticas necesarias para proporcionar la confianza adecuada en que un producto o servicio satisfará los requerimientos de calidad dados.

Eurométodo proporciona la ayuda necesaria para diseñar un buen plan de garantía de calidad. Las tareas relacionadas con esto se realizarán a lo largo de todo el proceso de adaptación de un sistema.

El Modelo de Productos de Eurométodo divide los productos en tres grandes grupos: Relativos al **dominio de intervención**, Relativos al **dominio del proyecto** y **Planes de entrega**. Y dentro de ellos, cada tipo de producto perfectamente identificado y definido. Es dentro de los productos relativos al dominio del proyecto donde se encuentran los productos de Garantía de Calidad. Concretamente los Planes y los Informes referidos a este área.

Haremos un recorrido por los productos propuestos por Eurométodo que se refieran a la Garantía de Calidad. Para ello se identificarán, bien productos propios de la garantía de calidad, bien aspectos referidos a la calidad de otros productos, que Eurométodo tiene en cuenta para su descripción.

En este punto estaremos en condiciones de poder aplicar a los productos identificados la información que proporciona la Guía para Establecer Puentes.

4.4.1. Productos Relativos al Dominio de Intervención

Son productos que describen el sistema de información, que pueden usarse como partes operacionales de él, o que facilitan el funcionamiento del mismo. En ellos se basan las decisiones de diseño de importancia contractual y que requieren la participación del cliente y del proveedor. Dentro de estos puede haber dos tipos:

*** Descripciones del sistema de información.**

Proporcionan conocimiento sobre el sistema considerado. Son productos tales como Requerimientos, Especificaciones y Prototipos. Ninguno de ellos son en sí mismos productos relacionados con la garantía de calidad. Estos productos se caracterizan basándose en ciertos aspectos, propiedades y propiedades no funcionales (éstas últimas solo en el caso de los Requerimientos) enfocados a caracterizar de forma completa el conocimiento sobre un sistema de información, que es necesario para facilitar una decisión concreta durante una adaptación.

Son las propiedades no funcionales las que se refieren al funcionamiento y calidad del sistema. Su identificación y descripción están basadas en el estándar ISO-9126 y algunas de ellas son:

-Costo, de producción, mantenimiento y operación.
-Beneficio, que aporta a la organización.
-Volumen, o número de ocurrencias de un cierto tipo.
-Frecuencia, de ejecución de un proceso, tarea o función.
-Duración, de un proceso, tarea o función.
-Eficiencia, o relación entre el nivel de funcionamiento y cantidad de recursos.
-Seguridad, o capacidad para prevenir accesos no autorizados.
-Criticalidad, o importancia que conlleva un fallo funcional.
-Exactitud, o capacidad para mantener su nivel de funcionamiento bajo ciertas condiciones dadas, durante un período de tiempo.
-Facilidad de mantenimiento, o esfuerzo requerido para hacer modificaciones.
-Portabilidad, o capacidad de ser transferido de un entorno a otro.
-Facilidad de uso.

*** Items operacionales.**

Pueden usarse como partes operacionales del sistema de información o facilitar su funcionamiento. Son productos tales como Dispositivos Hardware, Sistemas Software, Software de aplicación, Ocurrencias de Datos, Manuales de usuario y de operación, Instrucciones de trabajo, y Cursos de aprendizaje.

A todos los productos relativos al dominio de intervención se les atribuye un estado de calidad para asegurar que los procedimientos relacionados con la calidad de los mismos se completarán con éxito. Los estados de calidad de estos productos se caracterizan por Revisiones (para las descripciones) y Pruebas (para los items operacionales). En ambos casos, estos procedimientos se dirigen a demostrar la adecúación del producto a las estructuras y criterios de calidad definidos. Los procedimientos concretos a llevar a cabo dependerán de la adaptación particular y se prescriben en el Plan de garantía de calidad.

4.4.2. Productos Relativos al Dominio del Proyecto

Para facilitar las decisiones durante el proceso de producción el usuario necesita tener un conocimiento profundo de la producción del sistema de información del proveedor, para controlar la calidad de los productos relativos al dominio de intervención.

Pueden ser entregados por el proveedor para proporcionar el cliente control y confianza sobre el contenido y la calidad de los productos relativos al dominio de intervención. También pueden ser entregados por el cliente para definir sus requerimientos para la organización del proyecto.

Un producto relativo al dominio del proyecto captura conocimiento sobre los procesos internos del proyecto. Esto se caracteriza por las propiedades del proyecto que describen el producto. El rigor que se requiera en esto puede definirse por la selección de las propiedades del proyecto que prescriben los Planes y documentan los Informes. Hay dos tipos: Planes e informes.

*** Planes del proyecto (relativos a la gestión del S.I).**

Cliente y proveedor necesitan controlar el proceso de producción interna del proyecto. Esto se define en los Planes. También son un medio para definir la calidad necesaria con que deben construirse los productos relativos al dominio de intervención. Se caracterizan por un conjunto de propiedades que se refieren a las tres áreas de trabajo definidas en el Modelo de Transacción (Desarrollo, Garantía de calidad, Gestión de configuración), de las cuales nos ocuparemos únicamente de la Garantía de calidad.

Como hemos visto en el apartado anterior, para los productos relativos al dominio de intervención puede definirse un estado de calidad requerido. Este estado define los procedimientos de garantía de calidad que hay que llevar a cabo. La información relativa a esto es relevante para la relación cliente/proveedor.

El propósito de los Planes de garantía de calidad es definir los procedimientos que tiene que pasar un producto para ser considerado con una calidad adecuada. También prescribe los procedimientos de Revisión concretos para definir el estado de calidad que, junto con las propiedades, caracteriza los Planes de entrega y los otros Planes de proyecto. Su contenido se puede caracterizar por las siguientes propiedades:

-Con qué producto relativo al dominio de intervención está relacionado el plan.

-Organización de la gestión de calidad:
  -Estructura.
  -Tareas y responsabilidades.
  -Interfases.

-Definición de los procedimientos de garantía de calidad.
  -Referencia a los requerimientos de calidad que hay que chequear.
  -Referencia a los procedimientos de garantía de calidad que hay que aplicar para chequear los productos frente a los requerimientos.
  -Planificación de la ejecución de los procedimientos de garantía de calidad (recursos, fecha, ...).

*** Informes del proyecto (relativos a la producción del S.I.).**

En un producto, el cliente debe contar con los medios para valorar la adecuación de los productos relativos al dominio de intervención. Aparte de la valoración de su contenido, los clientes valoran si se reunen los requerimientos. Los informes capturan este conocimiento sobre el proceso de producción interna del proyecto. Indican cómo se han cumplido los planes. De la misma forma que los Planes, los Informes se caracterizan por medio de un conjunto de propiedades del proyecto que pueden estar referidas a cada una de las tres áreas de trabajo definidas por el Modelo de Transacción (Desarrollo, Garantía de Calidad y Gestión de Configuración). En este caso, además puede haber Informes sobre el Estado del Proyecto, que proporcionan una descripción del estado del proyecto con la mirada puesta en el Plan de entrega.

El propósito de los Informes de garantía de calidad es documentar que los procedimientos de garantía de calidad han sido completados con éxito. Su contenido se puede caracterizar por las siguientes propiedades:

-Con qué producto relativo al dominio de intervención, sometido a procedimientos de garantía de calidad, está relacionado el Informe.
-Informe sobre los procedimientos de garantía de calidad:
  -Estado de calidad que se requiere para ese producto en los puntos de decisión.
  -Definición o referencia a los requerimientos de calidad que fueron chequeados para el producto (que se han definido en el plan de garantía de calidad).
  -Definición o referencia a los procedimientos de garantía de calidad (que se han definido en el plan de garantía de calidad) que se han aplicado para chequear el producto frente a los requerimientos.
  -Información sobre la ejecución de la garantía de calidad (recursos involucrados, fecha, ...).
-Protocolo de los procedimientos de garantía de calidad:
  -Documentación de los resultados de los procedimientos de garantía de calidad llevados a cabo.
  -Procedimientos que se han adoptado para eliminar defectos identificados, si los hay; y resultados de los procedimientos de garantía de calidad relativos a esos procedimientos.

(Adicionalmente, una evaluación de la garantía de calidad puede identificar riesgos del proyecto).

-Evaluación de la garantía de calidad:
  -Pueden identificarse fuentes de defectos detectados.
  -Pueden identificarse desviaciones entre los resultados obtenidos y los esperados.
  -Puede proporcionarse una valoración del riesgo, basándose en los problemas identificados.
  -Pueden proponerse acciones correctoras.

4.4.3. Planes de Entrega

Durante el proceso de oferta, y en las transacciones cliente/proveedor referentes a la revisión de la planificación de la adaptación, los productos que se necesitan son documentos que definan la adaptación.

Los planes de entrega son los productos que definen la relación contractual cliente/proveedor (puntos de decisión y transacciones) durante los procesos de producción y completación. A través del uso de conceptos Eurométodo puede proporcionar unos medios de comunicación comunes.

Contienen los siguientes **elementos**:

-Descripción de la situación del problema:
  -Estado inicial del sistema de información.
  -Items operacionales de un sistema de información futuro, útiles para el proceso de producción.
  -Requerimientos de la adaptación: Descripciones útiles para el sistema de información, Plan de entrega, Planes del proyecto.
  -Estado final del sistema de información.
  -Items operacionales de un sistema de información que se produce como resultado de la adaptación y que el usuario quiere seguir usando después.
  -Planes para adaptaciones futuras.
-Razonamiento que lleva a la secuencia de puntos de decisión definidos en el Plan de entrega.
-Decisiones que se han tomado, papeles involucrados en las decisiones, transacciones que tienen lugar y productos que se intercambian.

Los Planes de entrega, en sí mismos, no son productos de garantía de calidad, aunque contienen a otros productos que sí lo son (Planes del proyecto). A la hora de describir la situación del problema tienen muy en cuenta la complejidad e incertidumbre del mismo, para prever riesgos y definir una estrategia de adaptación adecuada

5.Conclusiones

Buscamos un nexo metodológico entre Eurométodo y otras metodologías vía calidad del software. En la filosofía de Eurométodo hay un tema fundamental que es la mejora de la calidad de los sistemas que se obtengan. Este intento de mejorar la calidad está presente en todos los puntos de vista desde los que puede aplicarse. Sin embargo hemos elegido tres (Estrategia de Adaptación, Relación cliente/provedor y establecimiento de Puentes) para este estudio por ser los más representativos.

En la relación cliente/proveedor y en el establecimiento de Puentes se contemplan aspectos como complejidad e incertidumbre. Unas veces se trata de reducirlos para los sistemas que se van a obtener y otras para el proceso -en sí mismo- de obtener dichos sistemas. Pero es en la definición de la Estrategia de Adaptación donde complejidad e incertidumbre cobran una importancia vital: estos factores son la base para decidir de qué manera va a llevarse a cabo la adaptación. Se trata de obtener la forma idónea para llevar la adaptación, lo cual influirá en la calidad del sistema obtenido.

Complejidad e incertidumbre son factores que pueden evaluarse en muchos y muy diferentes aspectos de la adaptación de un sistema: Actores y relaciones entre ellos, información, procesos de la empresa, datos, funciones, interfases, especificaciones, requerimientos, entorno tecnológico, etc. agrupados por Eurométodo en Sistema de Información, Sistema de Computadora, Tareas del Proyecto, Estructura del Proyecto, Actores del Proyecto y Tecnología del Proyecto.

Complejidad e incertidumbre sirven para predecir riesgos. Riesgos en general, lo que ya es garantía de mejora en la calidad, y también riesgos sobre la calidad de los productos. Como vemos, complejidad e incertidumbre son factores que influyen notablemente en la calidad de los sistemas de información. También son factores importantes en la filosofía general de Eurométodo, y cruciales en uno de sus puntos esenciales . Podemos, por tanto, determinar la complejidad y la incertidumbre como un posible nexo metodológico entre Eurométodo y otros métodos, vía calidad.

6.Bibliografía y referencias

EUROMETODO, sus objetivos y desafíos. CIIBA (Comité Interministerial para la Informática y la Burótica en la Administración francesa). NOVATICA nº 107, pag. 25-31. Enero/Febrero 1994.
Foro español EUROMETODO. NOVATICA nº 107, pag. 32-33. Enero/Febrero 1994.
Marcelo J.; *Eurométrica.* NOVATICA nº 107, pag. 35-38. Enero/Febrero 1994.
EUROMETODO. El proyecto y los objetivos. MAP (Ministerio para las Administraciones Públicas). Marzo 1994.
Botella P. (Univ. Politéc. de Catalunya) y García G. (MAP); *El proyecto EUROMETHOD: situación actual.* Marzo 1994.
Overview. Euromethod, Version 0. Junio 1994.
Deliverable Model. Euromethod, Version 0. Junio 1994.
Transaction Model. Euromethod, Version 0. Junio 1994.
Method Bridging Guide. Euromethod, Version 0. Junio 1994.
Dictionary. Euromethod, Version 0. Junio 1994.
[Ahi] Ahituv N., Hadass M. and Neumann S.; *A Flexible Approach to Information System Development»*, MIS Quarterly, Junio 1984.
[Alt] Alter S. y Ginzberg, M.; *Managing Uncertainty in MIS Implementation*, Sloan Management Review, Fall, 1978.
[Boe91] Boehm B.; *Software Risk Management: Principles and Practices*, IEEE Software, Enero 1991.
[Brink] Brinkkemper S., Lange M. de, Looman R., Steen F. van der.; *On the derivation of method comparison by meta-modelling.* Case ´89. Third International Workshop on Computer-Aided Software Engineering. Imperial college, London. Julio 1989.
[Dav82] Davis Gordon B.; *Strategies for Information Requeriments Determination*, IBM Systems Journal, vol.21, Nº1, 1982.
[DoD] Department of Defence, United States, Military Standard Software Development and Documentation (draft). MIL-STD-SDD (DRAFT), 22 Diciembre 1992.
[Gib] Gibson C.F., Singer C.J., Schnidmann A.A. and Davenport T.H.; *Strategies for Making an Information System Fit Your Organisation.* Management Review, Enero 1984.
[ISO-9126] ISO 9126 Information technology - Software product evaluation - Quality characteristics and guidelines for thir use: 1991.
[Mat] Mathiassen L. and Stage J.; *The principle of limited reduction in software design.* Information Technology and People. 6:2-3 1992. Norhwind Publication.
[McF82] McFarlan W.; *Portfolio Approach to Information Systems.* Journal of Systems Management, Enero 1982.
[Mor] Morley C.; *Méthodes et contingences.* Thèse de doctorat. Febrero 1991.
[Rob] Robinson K.; Lower ISE Guides, CCTA internal document.
[Rop] Ropponen J.; *Risk management in Information System Development.* Licentiate thesis in Information Systems. Mayo 1993.
[Saar] Saarinen T.; *Success of Information System Investments -Contingent strategies for development and a multidimensional approach for evaluation.* Draft version of the doctoral dissertation, Abril 1992.
[VM92] V-Model. German Federal Ministry of the Interior, Agosto 1992.

Ingeniería del Software

Reutilización del software en el sector espacial

Fernando Aldea Montero
CRISA, Sección de Calidad del Software, P.T.M.
Torres Quevedo, 9. 28760 - TRES CANTOS (MADRID)
Tlfno: +34-1-8032728; Fax: +34-1-8036528
Gabriel Sánchez Gutiérrez
SEMA-GROUP SAE - Investigación y Desarrollo
c/ Albarracín, 25. 28037 - MADRID
Tlfno: +34-1-3272828; Fax: +34-1-7543252

Resumen: *este trabajo presenta algunos resultados preliminares del experimento* ***IRIS (Improving Reuse in Space)****, que pretende ayudar a identificar posibles mejoras en el ciclo de vida habitual en los proyectos espaciales, dirigidas a aumentar el nivel de reutilización de componentes software sin disminuir su calidad.*

1. Introducción (contexto de trabajo en CRISA)

CRISA desarrolla sistemas para equipos embarcados, que contienen hardware y software. Se trata normalmente de software empotrado de tiempo real, con unos requisitos muy exigentes de rendimiento y a veces de memoria. En consecuencia, y aunque la mayor parte del código es escrito en lenguaje Ada, no es extraño tener que recurrir a lenguajes ensambladores para algunas partes críticas.

Normalmente se utiliza un entorno mixto compuesto por herramientas CASE de análisis y diseño, compilador cruzado de Ada, herramientas de soporte (análisis estático y dinámico del código, control de versiones), emulador del microprocesador, analizador de protocolos, etc. En este sentido, se detecta la carencia en el mercado de un entorno integrado válido para este tipo de desarrollos cruzados. Últimamente se ha empezado a experimentar con CONCERTO-ESSDE (recomendado por la **ESA**), un entorno que incluye conexiones o herramientas propias para casi todas las actividades del ciclo de vida software.

El proceso de desarrollo normalmente sigue los estándares de la **ESA** (**Agencia Europea del Espacio**) **[1]**, desdoblados en un desarrollo incremental de dos o más versiones, en las cuales se va aumentando la funcionalidad del software hasta cubrir toda la especificación. Esto es necesario para coordinar el proceso de desarrollo software con los sucesivos modelos hardware asociados a un equipo (modelo de ingeniería, modelo de calificación y modelo de vuelo).

2. Planteamiento del experimento

El experimento consiste en el desarrollo de una aplicación espacial típica siguiendo dos caminos paralelos. En primer lugar, un desarrollo nominal, en el cual se respeta el típico ciclo de vida 'en cascada' (modelo de Waterfall), sin prestar especial atención a los aspectos relacionados con la reutilización. Este primer desarrollo es un proyecto externo contratado con la ESA, sobre el cual no puede realizarse ningún experimento que suponga modificaciones a los requisitos originales.

En segundo lugar, se realiza un desarrollo orientado a la reutilización, durante el cual un subconjunto del diseño software existente es refinado de acuerdo a una estrategia de 'desarrollo PARA reutilización'. Esto conlleva la identificación de aquellos componentes potencialmente reutilizables, y el desarrollo de los mismos siguiendo criterios orientados a incrementar el potencial de reutilización.

En ambos casos se han especificado unos requisitos de calidad determinados, coherentes con los objetivos del proyecto. Requisitos (factores) traducidos en una serie de criterios y métricas que han configurado un modelo de calidad y otro de reusabilidad apropiados para este dominio.

En estos modelos se identifica un conjunto de métricas que deben ser recogidas durante el desarrollo de ambos proyectos, y al final servirán para realizar un estudio comparativo con un doble objetivo:

- Identificar qué mejoras relacionadas con la calidad y la reutilización se obtuvieron en el segundo desarrollo (estrategia de 'desarrollo PARA reutilización'), y cuánto esfuerzo se necesitó para conseguirlas.
- Estimar el esfuerzo que se ahorrará en el futuro, cuando estos componentes sean aprovechados en una estrategia de 'desarrollo CON reutilización'.

En el contexto de este experimento, hemos distinguido dos tipos de componentes software: *componentes del tipo 1*, formados por requisitos, diseño, código fuente, ficheros de pruebas y documentación de usuario; y *componentes del tipo 2*, formados únicamente por el código fuente y la documentación de usuario (incluyendo posiblemente alguna documentación de pruebas). De esta manera, se admite la posibilidad de reutilizar un componente, o bien al completo (tipo 1), o bien sólo su implementación (tipo 2).

Los dos desarrollos se realizan utilizando **HOOD** como metodología de diseño, y **Ada** como lenguaje de programación. Ambos han sido seleccionados por su difusión en los programas de la ESA, y por su adecuación al objetivo de reutilización.

HOOD es un método de descomposición jerárquica de un diseño software basado en objetos. Proporciona mecanismos básicos para describir el problema a resolver, y para representarlo gráficamente y formalizarlo en un lenguaje muy próximo al Ada.

Ada es un lenguaje de programación orientado a tiempo real y a grandes sistemas, que proporciona diversas facilidades para desarrollar componentes reutilizables: unidades genéricas, paquetes, ocultamiento de datos, tareas de alto nivel, y posibilidades para aislar la dependencia del hardware en una unidad o módulo.

4. Breve descripción de las dos aplicaciones

4.1.Desarrollo nominal (ICE software)

La **ICE (Electrónica de Control del Instrumento)** es una parte del subsistema de dirección y control del instrumento MIPAS, que incluye software embarcado y empotrado dentro del hardware. Las principales funciones de este software son:
- dirigir y controlar la ICE (secuencia de arranque, apagado de emergencia);
- dirigir las transiciones de modos de acuerdo a las secuencias operacionales;
- comunicar con el ordenador central a través de telemetría y macrocomandos;
- mantener el área histórico (almacenamiento temporal de eventos y errores);
- secuenciamiento de las tomas de datos;
- procesamiento de otros subsistemas.

La ICE implementa la mayoría de las funciones típicas de toda unidad de control de un instrumento: gestión de modos, adquisición y monitorización de datos, telemetría y macrocomandos, etc. Funciones específicas con respecto a otras unidades de control son todas las secuencias asociadas a cada transición de modo, o al control de cada subsistema.

4.2. Desarrollo paralelo orientado a la reutilización

Se ha seleccionado un subconjunto de las funciones típicas ya comentadas: adquisición de datos, monitorización y telemetría. Todas ellas presentan un gran potencial de reutilización, dentro de los programas de la ESA.

El subsistema resultante puede ser descrito como sigue. El software adquiere a intervalos regulares diversos parámetros sobre el estado del hardware, y los controla para determinar si el sistema está operando correctamente. Es posible configurar a través de una orden externa qué parámetros se deben adquirir, con qué frecuencia, y con respecto a qué límites deben ser controlados. Todos los eventos significativos (internos o externos) deben ser comunicados al ordenador central de a bordo, para lo cual es necesario crear diversos formatos de telemetría. Algunos eventos son transmitidos en el momento que ocurren, otros son almacenados temporalmente en el área histórico, y después transmitidos regularmente.

5. Metodologías de soporte

Para la realización del experimento IRIS, se ha decidido utilizar como base la metodología propuesta por **REBOOT [6]**, completada por algunos aspectos de los estándares de la OTAN para reutilización del software [3-5].

REBOOT propone un camino para la introducción sistemática de la reutilización software en una organización, dividido en dos etapas:
- desarrollo PARA la reutilización, que consiste en invertir más esfuerzo en el desarrollo de algunos componentes con la intención de reutilizarlos en proyectos posteriores;
- y desarrollo CON reutilización, o sea recuperar en otros proyectos el esfuerzo invertido, a través de la reutilización de los componentes previamente desarrollados.

REBOOT también ofrece un conjunto de herramientas de soporte para la puesta en práctica de la metodología, como una solución integrada dentro del entorno **CONCERTO-ESSDE**.

CONCERTO es un entorno de herramientas comercializado por SEMA-GROUP, que fue concebido para el desarrollo de aplicaciones técnicas grandes, y que recientemente ha sido recomendado por la ESA para proyectos espaciales. CONCERTO incluye herramientas generales para el control del desarrollo software (modelización de procesos, gestión de la configuración, ayuda a la trazabilidad), y específicas que soportan distintos lenguajes y metodologías de diseño (HOOD, Ada, C++, ingeniería inversa de Ada a HOOD, etc.).

La metodología REBOOT permite la inclusión de las características del dominio del usuario en lo que a calidad y clasificación se refiere. Primero hay que definir esquemas de clasificación de componentes apropiados al dominio de software embarcado. También hay que definir modelos de calificación adaptados a los niveles de calidad requeridos por la ESA para software embarcado. Finalmente, REBOOT suministra unas recomendaciones para cada fase del desarrollo, las cuales deben ser adaptadas (por ejemplo, recomendaciones específicas para HOOD).

	NIVEL				
FACTOR	**MODELO DE CALIDAD (ICE SOFTWARE)**	**MODELO DE CALIDAD (REBOOT)**	**MODELOS DE REUSABILIDAD (REBOOT)**		
			Como está	***Pequeños cambios***	***Cambios importantes***
CORRECCIÓN	alto	alto	alto	alto	alto
EFICIENCIA	alto	bajo	alto	bajo	bajo
FIABILIDAD	alto	alto	alto	normal	alto
FLEXIBILIDAD	bajo	normal	normal	normal	alto
FAC. DE EXPANSIÓN					
FAC.MANTENIMIENTO	alto	alto	alto	normal	normal
FAC. PRUEBA	alto	normal	normal	normal	alto
PORTABILIDAD	muy bajo	normal	muy alto	muy alto	alto
INTEROPERABILIDAD					

TABLA I

6. Modelos de calificación para el experimento

Se ha definido un modelo de calidad y otro de reusabilidad para este experimento. Este último, además, se ha desdoblado en tres modelos, según el grado de reutilización que se pretenda alcanzar: *como está* (reutilización de un componente sin cambios),*con pequeños cambios* (reutilización tras pequeñas adaptaciones del código fuente) o *con cambios importantes* (reutilización tras cambios en la arquitectura del componente).

Un modelo de calificación se estructura en tres niveles de arriba-abajo, de acuerdo al esquema clásico de McCall factor-criterio-métrica (F-C-M)[7]. Cada factor se relaciona con uno o más criterios, cada uno de los cuales se relaciona a su vez con varias métricas. Un criterio puede estar relacionado con dos o más factores, así como una métrica puede afectar a dos o más criterios.

Para decidir el nivel de los factores en cada uno de los modelos, se ha seguido un método de evaluación cualitativo propuesto por la ESA[2]. En la **tabla I** se muestra una comparación entre los resultados obtenidos para el desarrollo nominal, y para los modelos de calidad y reusabilidad de REBOOT seleccionados.

Las principales diferencias entre los modelos de calidad y reusabilidad se encuentran en los siguientes factores:

Eficiencia: los requisitos de rendimiento y memoria son especialmente exigentes en el proyecto ICE software. Normalmente no serán tan altos, y la eficiencia será penalizada en favor de otros factores de calidad.

Flexibilidad:un componente reutilizable debe permitir nuevas funcionalidades sin un gran esfuerzo.

Portabilidad e interoperabilidad:un componente reutilizable debe ser fácilmente adaptable a otros entornos hardware o sistemas operativos.

5.1. Selección de las métricas

Las métricas elegidas se clasifican en dos grupos, según la forma de recogerlas: *métricas de cuestionario* y *métricas automáticas*. En cuanto a las primeras, se ha definido un cuestionario para cada uno de los productos asociados a un componente (especificación, diseño, codificación, pruebas unitarias y documentación de usuario). Los cuestionarios son contestados de forma manual, bien por el diseñador del componente, bien por un ingeniero de calidad. Se ha tratado de no superar las treinta preguntas entre todos los cuestionarios (a costa de perder información).

El modo de llegar a una respuesta en cada pregunta es 'público' (lo cual no quiere decir 'objetivo'), en el sentido de que en todo momento se puede saber qué pasos siguió el encuestador hasta alcanzar una conclusión. Lo que no es posible es eliminar la carga subjetiva que casi todas las preguntas tienen, según quien las evalúe.

Las métricas automáticas se han seleccionado a partir de las métricas básicas proporcionadas por la herramienta AUDIT de CONCERTO, las cuales se clasifican en seis grupos: métricas del grafo de control, métricas de Halstead, métricas de complejidad léxica, métricas de complejidad de las estructuras de datos, métricas de complejidad de las interfaces, y normas de programación (específicas para el lenguaje Ada). Algunas de estas métricas básicas se han combinado para generar otras métricas más significativas. Se han definido límites de aceptación para cada una de ellas, bien basándose en la experiencia de CRISA en otros proyectos, bien por la correlación observada entre dos o más métricas.

5.2.Normalización de las métricas

Las métricas han sido normalizadas, en el sentido de que todas toman valores comprendidos entre 0 y 1, donde un valor próximo al 0 significa una mala puntuación, mientras que un valor cercano al 1 significa una buena puntuación. Desde este punto de vista, se han clasificado las métricas en cuatro tipos:

- ***Métricas discretas,***que sólo toman los valores 0 y 1. Ejemplo: número de instrucciones 'goto' présentes en un componente (no debe haber ninguna).
- ***Métricas del valor límite,*** caracterizadas por un rango de valores aceptables desde el 0 hasta un cierto límite, a partir del cual los valores no son aceptables. Ejemplo: la complejidad ciclomática de un subprograma (valores por debajo de 10-12 son todos aceptables).
- ***Métricas del valor óptimo,*** caracterizadas por un rango de valores aceptables en torno a un valor óptimo, mientras que los valores que se alejan demasiado de dicho valor óptimo (tanto por defecto como por exceso) no son aceptables. Ejemplo: el porcentaje de líneas de comentarios de un componente (es tan malo que un componente esté poco comentado, como que lo esté en exceso).
- ***Métricas de dependencia lineal,*** cuyo valor es mejor cuanto más próximo al 0 se encuentre. Ejemplo: número de variables globales del componente (aún aceptando que pueda haber alguna, cuantas menos mejor).

La **figura 1**muestra las gráficas asociadas a estos métodos:

5.3. Ponderación de los modelos de calificación

Una vez determinada la composición 'cualitativa' de los modelos siguiendo un proceso de arriba abajo (estructura F-C-M), se ha determinado su composición 'cuantitativa', en un proceso de abajo arriba:

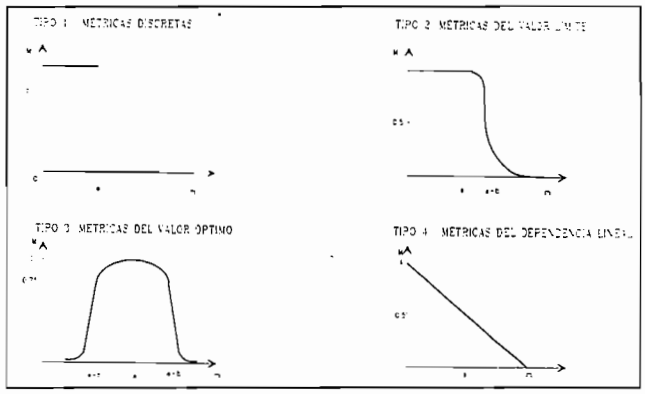


Figura 1: Métodos de Normalización de Métricas

1) Partiendo de las métricas ya normalizadas, para cada criterio se ha ponderado la importancia de cada métrica que lo determina.
2) A su vez, en cada factor se han ponderado los criterios que lo determinan.
3) Finalmente, en cada modelo de calificación se ha ponderado la importancia de cada factor seleccionado, de acuerdo al nivel requerido.

La **tabla II** muestra un ejemplo completo de este proceso (métricas —>criterios —>factores —>modelo).

7. Primeras conclusiones

Si bien el estado de avance de los dos desarrollos no permite aún obtener grandes conclusiones, sí que se pueden mencionar algunas lecciones aprendidas:

- En este tipo de dominios es muy difícil obtener componentes reutilizables en los niveles altos de la arquitectura. Hay que optar entre parametrizar demasiadas características del componente (a costa de hacer más compleja su instanciación), o bien limitar el potencial de reutilización fijando a priori algunas de esas características. En este experimento, lo que se ha intentado en algunos casos es crear un nivel intermedio en la descomposición del componente, de tal manera que por debajo el software deberá ser modificado para cada entorno hardware específico, y por encima no.
- Es difícil alcanzar un compromiso entre generalidad y rendimiento de las componentes reutilizables. Un buen número de características de HOOD a Ada orientadas a favorecer la reutilización penalizan el consumo de tiempo y el de memoria. Esta circunstancia, que no tendría tanta importancia en otros dominios, obliga en el caso de software embarcado de tiempo real a identificar claramente qué modificaciones se deben hacer en un componente (a la hora de instanciarlo) para ahorrar tiempo de ejecución y memoria.
- En general, las métricas de cuestionario tienen un peso muy superior a las métricas automáticas (del orden de 80-20), lo cual incrementa los recursos necesarios (mediciones manuales), y disminuye la objetividad. El problema está en parte en las herramientas disponibles en el mercado, y en parte en el propio estado del arte.
- Se ha constatado la tremenda dificultad de fijar límites y pesos en los modelos de calificación; sacando dos conclusiones:
  . Necesidad de contrastar los límites establecidos con datos reales. Por ejemplo, si se ha decidido que una complejidad ciclomática media superior a 10 es un indicador de fracaso, hay que validar esta hipótesis midiendo la tasa de fallos del componente en cuestión.
  . Conveniencia de que un experto de CRISA realice una evaluación independiente de cada componente. Si sus conclusiones difieren sustancialmente de las obtenidas con los modelos, los pesos deberían ser revisados, hasta llegar a unos modelos estables y realistas.

7. Trabajo futuro

Ambos desarrollos han superado la fase de diseño arquitectural, y se encuentran en fase de producción (diseño detallado, codificación y pruebas unitarias). Una vez completada la misma, comenzará el estudio comparativo entre ambos conjuntos de componentes, basándose en las métricas recogidas hasta ese momento. Al mismo tiempo, se utlizarán los modelos de calificación definidos en REBOOT para evaluar la calidad y la reusabilidad de los componentes desarrollados. Según los resultados de esta evaluación, se decidirá si se puede establecer una librería inicial de componentes reutilizables en futuros proyectos.

8. Referencias

[1] **ESA PSS-05-0;** *Software Engineering Standards*. Edición 2. Febrero de 1991.
[2] **ESA PSS-01-220**; *Software Quality Assurance*. Borrador 1.8, Octubre de 1991.
[3] *NATO standard for the development of reusable software components.*
[4] *NATO standard for management of a reusable software component library.*
[5] *NATO standard for software reuse procedures.*
[6] *Software Reuse: A Holistic Approach.* Even-André Karlsson. Wiley. 1995.
[7] *Factors in Software Quality Assurance*. RADC-TR-77-369. Rome Air Development Center, n.p. McCall, Richards and Waiters. Noviembre de 1977.

CRITERIO "NIVEL DE AUTO-DESCRIPCIÓN		FACTOR FLEXIBILIDAD		MODELO-REUSABILIDAD 'COMO ESTÁ'	
Métrica	**Peso**	**Criterio**	**Peso**	**Factor**	**Peso**
Porcentaje de comentarios	0,05	Modularidad	0,30	Correción	0,15
No utilización/tipos anónimos	0,05	Simplicidad	0,20	Eficiencia	0,15
No utilización instrucción 'use'	0,10	Nivel de autodescripción	0,20	Fiabilidad	0,15
Prohibición de constantes literales	0,10	Generalidad	0,15	Flexibilidad	0,10
Longitud media de identificadores	0,05	Nivel de documentación	0,15	Facilidad de mantenimiento	0,15
Código bien comentado	0,20			Fac. de prueba	0,10
Uso de autodescriptivos	0,25			Portabilidad	0,20
Presentación de código fuente	0,20				
TOTAL	***1,00***	***TOTAL***	***1,00***	***TOTAL***	***1,00***

TABLA II

Ingeniería del Software

Modelado y soporte del proceso software. El entorno DELPHOS

Juan Carlos Yelmo García
Departamento de Ingeniería de Sistemas Telemáticos
ETSI Telecomunicación, Ciudad Universitaria s/n,
28040 Madrid. //yelmogar@dit.upm.es

Este trabajo ha sido parcialmente financiado por la Comisión Interministerial de Ciencia y Tecnología (CICYT) a través del proyecto EMEDAS. Delphos es el nombre de la ciudad donde se localizaba el más célebre oráculo de la Grecia antigua. Se dice que a él acudían gobernantes y sabios a conocer los designios de los dioses, designios tan oscuros e inestricables que precisaban de la interpretación de la Pitiao sacerdotisa del templo.

Palabras clave: *Ingeniería del software, Modelado del proceso software, Técnicas de Descripción Formal, Orientación a objetos, Plataformas de objetos distribuidos.*

Resumen: *Se ofrece una introducción y estado del arte del modelado del procesos como disciplina emergente que está encontrando su aplicación en el campo de la ingeniería del software con énfasis en su aplicación al desarrollo de nuevas arquitecturas de entornos CASE denominadas en-tornos basados en proceso. En este sentido se ofrece el con-texto técnico de los entornos integrados y las iniciativas de estandarización relacionadas, para finalizar con la expe-riencia de desarrollo del prototipo DELPHOS como marco de integración sobre plataforma de proceso distribuido.*

1. Introducción

El presente artículo ofrece una introducción y estado del arte del modelado del procesos como disciplina emergente que está encontrando su aplicación en el campo de la ingeniería del software. Esta disciplina permite una descripción rigurosa y, en algunos casos, susceptible de tratamiento automatizado, de los procesos de desarrollo software. Este campo no es sin embargo el Único en que dicha disciplina está encontrando aplicación. Podemos citar entre otros los siguientes: modelado de organizaciones empresariales, reingeniería de procesos empresariales, ingeniería concurrente, manufactura integrada mediante ordenador (CIM), etc.

En la siguiente sección se presenta una introducción a los conceptos básicos, ventajas y campos de aplicación y principales formalismos de modelado citados en la literatura expuestos de manera comparativa. La sección termina poniendo en relación el modelado del proceso software con el modelo de ciclo de vida representado por el paradigma espiral para desarrollo software.

En la sección 3 se introducen los conceptos fundamentales e iniciativas de estandarización relacionados con la integración de herramientas en entornos de desarrollo (entornos CASE) para, a continuación, ponerlos en relación con el modelado del proceso software para introducir los llamados entornos basados en proceso.

La sección 4 presenta una aplicación de las disciplinas anteriormente mencionadas al desarrollo de un entorno basado en proceso para desarrollo de aplicaciones de objetos distribuidos en el campo de las telecomunicaciones. Dicho entorno, llamado DELPHOS, representa un enfoque basado en formalismos gráficos, Redes de Petri y la plataforma de objetos distribuidos OMG-CORBA, para utilizar los conceptos de modelado de proceso en la implementación de un marco de integración distribuido basado en proceso.

El artículo termina con las conclusiones y resultados obtenidos hasta la fecha junto con las principales líneas de investigación abiertas por el trabajo aquí presentado.

2 Modelado del proceso software

Desde que se acuñó el concepto de ingeniería software se han propuesto y ensayado infinidad de modelos de ciclo de vida de desarrollo de productos software. Los modelos en cascada, evolutivo, transformacional, espiral, etc. representan paradigmas conceptuales que, junto con elementos de proceso y organizacionales *ad hoc,* se han utilizado hasta ahora para el desarrollo de software en todos los dominios de aplicación.

De un tiempo a esta parte estan cobrando actualidad trabajos encaminados a modelar el proceso de desarrollo en sí y a desarrollar lenguajes o medios gráficos que permitan describir de forma rigurosa el proceso de desarrollo. En este sentido, se han propuesto como alternativa a la descripción narrativa clásica de los procesos de desarrollo distintos enfoques de formalización, entre los que cabe destacar el uso de lenguajes de programación [Oste87], herramientas de análisis/diseño de sistemas utilizadas para el modelado del proceso software [Kell89] o el desarrollo de lenguajes específicos para modelado de procesos [Sa92]. En el caso de utilización de lenguajes de programación (generales o específicos), la actividad de desarrollo sistemático del proceso software se conoce como ***Programación de Proceso.*** En opinion de algunos autores [Oste87], es deseable que el lenguaje de codificación del proceso sea próximo, si no idéntico, a los lenguajes utilizados en el desarrollo de los propios productos software. Esto permitiría al entorno y su conjunto de herramientas soportar el desarrollo tanto de procesos como de productos. Sin embargo, no siempre el formalismo de desarrollo presenta las características adecuadas (concurrencia, persistencia, etc.) o capacidad expresiva para la formalización y ejecución de modelos de proceso.

Potencialmente, pueden derivarse muchas ventajas del desarrollo de descripciones rigurosas del proceso software. Estas descripciones proporcionarían la base para mejorar visibilidad, comunicación y coordinación en los procesos software y, si son suficientemente rigurosas, la capacidad de análisis y detección de errores imprescindibles para mejorar los procesos de desarrollo. Además se forzaría la utilización de un modelo de desarrollo concreto mediante el control de interacción de las herramientas del entorno.

Kellner ([Kell89]) propone el siguiente conjunto de características deseables en cualquier formalismo/metodología de modelado de procesos:

- Disponibilidad de un lenguaje visual para representación de la información.
- Soporte a mÚltiples ***vistas*** complementarias del proceso. Algunas de las perspectivas Útiles en la descripción de los procesos podrían ser:
  - ***Funcional.*** Actividades a realizar.
  - ***Comportamiento.*** Cuándo y cómo se realizarán las ta-

reas (*timing*, realimentación, paralelismo, precedencia, etc.).
- ***Organización.*** Describe qué agentes implementan las tar-eas y en que unidad de la organización se realiza el trabajo.
- ***Información***. Visión global y abstracta de los datos relevantes del modelo de proceso y de los objetos producidos. Estos datos permitirían la evaluación cuantitativa del proceso (esfuerzos, métricas, simulación, etc.).

- Soporte a diferentes niveles de abstracción para cada una de las perspectivas del punto anterior.
- Sintaxis y semántica formalmente definidas con objeto de obtener modelos computables.
- Capacidad de análisis (completitud, consistencia, etc.).
- Posibilidad de simulación del comportamiento del proceso directamente desde su descripción.
- El modelo debe poder tomar un papel activo en la ejecución del proceso, representando una guía de la ejecución y manteniendo un histórico de los pasos de proceso ejecutados.
- Ofrecer un entorno automatizado de configuración, llamada y ejecución de herramientas.

La **semántica** de un lenguaje de descripción de procesos debe ser adecuada para describir los siguientes elementos:
- **Productos**: Elementos de información identificables. Describibles por descomposición en tipos primitivos o mediante un sistema de clases (herencia, especialización, etc.).
- **Actividades**: Transformación de entradas en salidas. Describibles en términos de entradas, salidas, precondiciones y postcondiciones.
- **Agentes**: Elementos que realizan actividades. Estos serán elementos informáticos o humanos y, en general, podrán tener diferentes *vistas* del proceso.
- **Comunicación**: Sincronización de actividades y transferencia de productos entre agentes.
- **Decisiones**: Elecciones inspiradas por los objetivos del desarrollo.

Para terminar esta breve introducción a las notaciones de modelado de procesos, querríamos señalar aquí los principales objetivos y ventajas del modelado formal del proceso software en relación con sus diferentes áreas de aplicación.
- **Facilitar el entendimiento humano y la comunicación**. Permitiendo que el grupo de desarrollo comparta un formato común de representación con información suficiente para facilitar que un individuo o grupo de trabajo lleve a cabo el proceso modelado.
- **Proporcionar soporte al proceso de mejora y evolución del proceso software.** Identificando los componentes necesarios de un proceso de desarrollo software de alto rendimiento para reutilizar elementos de proceso efectivos y bien definidos en futuros proyectos, comparar procesos de desarrollo alternativos y soportar la evolución gestionada del proceso.
- **Proporcionar soporte a la gestión del proceso de desarrollo.** Con medios para monitorizar, gestionar y coordinar el proceso, así como proporcionar la base para la definición y aplicación de métricas del proceso.
- **Guiar la integración herramientas de soporte que proporcionen directrices en la ejecución del proceso software.** En definitiva, proporcionando la base para la contrucción de un entorno de desarrollo efectivo. Esta es la aplicación más interesante desde el punto de vista de este artículo.
- **Soportar la ejecución automatizada de elementos del pro-ceso.** Automatizando porciones de proceso (subprocesos) que se realizarían sin intervención humana.

2.1. Paradigmas para modelado de procesos

Los diferentes lenguajes y representaciones para modelado de procesos pueden evaluarse en la medida en que proporcio-nan construcciones Útiles para representar el proceso software y permitir el razonamiento sobre los diferentes aspectos de un proceso. En este sentido, los lenguajes de programación y especificación software proporcionan un medio que ofrece un marco conceptual adecuado y permite representar y ejecutar un proceso computacional. Es por esto que la mayoría de los investigadores en este campo han comenzado por ensayar este tipo de notaciones para modelar el proceso software. Hay referencias de utilización de la mayoría de las familias de lenguajes de programación y especificación para este propósito. La siguiente lista ofrece algunos ejemplos representativos de los tipos de lenguajes utilizados [Curt92], junto con las referencias bibliográficas de los más interesantes desde nuestro punto de vista.
- Lenguajes de programación procedimentales: APPL/A *(Ada Process Programming language based on Aspen)* [Sutt90]
- Notaciones para análisis y diseño de sistemas: STATEMATE [Hare90]
- Lenguajes y enfoques de inteligencia artificial: GRAPPLE [Huff89], AP5, MARVEL, MVP
- Sistemas basados en eventos y disparos: AP5, APPL/A, STATEMATE
- Máquinas de transición de estados y redes de Petri: Redes de Interacción de roles (RIN) [Sing92], STATEMATE, SPADE [Band93]
- Control de flujo: MVP
- Lenguajes funcionales: HFSP [Suzu91]
- Notaciones formales: Gramáticas de contexto libre
- Lenguajes de modelado de datos: APPL/A, PMDB, STATEMATE
- Notaciones de modelado de objetos: AP5, MARVEL, MVP
- Redes de precedencia: SPMS
- Modelado cuantitativo: Dinámica de sistemas

La **tabla 1** ofrece una visión de la medida en que los anteriores enfoques para modelado de procesos cubren las diferentes perspectivas básicas presentadas en la sección anterior (funcional, comportamiento, organización e información). Esta tabla junto con la relación anterior ofrece una panorámica indicativa del tipo de notaciones y lenguajes utilizados para modelado de procesos hasta el momento actual.

Para completar este sucinto estado del arte se ofrece a continuación un resumen de cuatro de los enfoques más extensamente utilizados incluyendo ejemplos concretos de lenguajes utilizados: ***modelos de programación, modelos funcionales, modelos basados en planes y modelos basados en redes de Petri.***

Tipo	funcional	comport.	organiza.	informac
Leng.procedimentales	*	*		*
Notaciones anál/diseño	*		*	*
Lenguajes IA	*	*		
Eventos y disparos		*		
FSM y PN	(RIN)	*	(RIN)	
Control flujo		*		
Lenguajes funcionales	*			
Notaciones formales	*			
Modelado datos				*
Modelado objetos			*	*
Redes precedencia		*		

Tabla 1: Formalismos de modelado y perspectivas cubiertas

2.1.1 Modelos de programación

En su ya clásico artículo *"Software processes are software too"* [Oste87], Osterweil sostiene que, puesto que la especificación de un proceso es una forma de programación, los procesos pueden modelarse utilizando las técnicas y herramientas que ya son familiares para los programadores. En este sentido, las descripciones de proceso pueden modelarse como algoritmos y ser analizadas como tales. El mismo autor puso en práctica este enfoque al desarrollar junto con sus colegas el lenguaje APPL/A para desarrollo de programas de proceso ejecutables en un ordenador.

APPL/A es una extensión de Ada basada en Aspen, un modelo de datos del estilo de los diagramas entidad-relación. Respecto de Ada, APPL/A añade la representación explícita de relaciones persistentes entre objetos software, operadores de disparo para propagar actualizaciones a través de estas relaciones y predicados para representar restricciones asociadas a las relaciones. APPL/A no representa un enfoque puramente procedimental al admitir la posibilidad de incluir descripciones declarativas, lo que le permite soportar varios paradigmas de representación. El modelado realizado con lenguajes como APPL/A es muy procedimental y conduce a modelos de control similar al proporcionado por entornos de desarrollo software convencionales.

2.1.2 Modelos funcionales

Podemos describir esta familia de notaciones a través de un ejemplo representativo como es HFSP *(Hierarchical and Functional Software Process).* Este es un lenguaje fundamentalmente declarativo en el que las trazas de ejecución pueden mostrarse en forma de diagramas. En HFSP un proceso se representa como una colección de elementos de proceso con atributos de entrada y salida. Más especificamente, un proceso se expresa como un conjunto de funciones matemáticas que representan la relación entre las entradas y salidas del elemento de proceso. Cada una de estas funciones es a su vez jerárquicamente descomponible en subelementos de proceso, donde los atributos de entrada/salida del elemento ***padre*** deben ser satisfechos por los correspondientes atributos de sus subelementos ***hijos***. Esta descomposición continÚa hasta que se generan pasos de proceso que se pueden hacer corresponder con invocaciones a las herramientas correspondientes o a operaciones manuales. En general, en HFSP los procesos pueden ejecutar concurrentemente, limitados Únicamente por la disponibilidad de sus atributos de entrada. Sin embargo, HFSP cuenta con mecanismos para representar otras estructuras de ordenación como la iteración, la secuenciación o la sincronización; así como con elementos de control dinámico del comportamiento como creación, destrucción, suspensión y reanudación de elementos de proceso. En resumen, HFSP está principalmente orientado a cubrir las perspectivas funcional y de comportamiento, con algún soporte para la perspectiva informacional.

2.1.3 Modelos basados en planes

Este enfoque sugiere que uno de los grandes problemas que conlleva la ejecución de modelos de proceso proviene de la dificultad de modelar todas las posibles contingencias presentes en el mundo real. La solución a este problema es especialmente difícil desde un enfoque procedimental, puesto que una representación procedimental carece de elementos que representen el bagaje técnico y los elementos de razonamiento que han llevado a elegir una determinada opción de representación. Por tanto, el marco de razonamiento subyacente a la definición de un proceso esta ausente del modelo, impidiendo razonar sobre las opciones de ejecución en función de las condiciones presentes en cada estado de la ejecución del proceso. En este sentido, este enfoque sugiere que este problema puede tratarse de forma más efectiva desde el paradigma de planificación surgido del campo de la Inteligencia Artificial, donde los operadores que representan las acciones posibles en un proceso se seleccionen en función de la satisfacción de sus precondiciones. Estos operadores se aplican al estado en que se encuentre el proceso en ejecución con objeto de llevar el estado del proceso más cerca del objetivo deseado.Huff y Lesser [Huff89] han desarrollado un lenguaje basado en restricciones llamado GRAPPLE que modela el desarrollo software como un conjunto de objetivos, subobjetivos, precondiciones, restricciones y efectos. Con GRAPPLE se construyen modelos de proceso a partir de dos componentes fundamentales: un conjunto de pasos de proceso y un conjunto de restricciones sobre cómo los pasos de proceso pueden seleccionarse, ordenarse y aplicarse. El éxito en la aplicación de sistemas basados en restricciones y planifi-cación como GRAPPLE depende del éxito de sus diseñadores al codificar el conocimiento sobre el entorno y la jerarquía de objetivos en función de componentes del lenguaje.

2.1.4. Modelos basados en redes de Petri

Una perspectiva sobre los proyectos de desarrollo software que resulta ortogonal respecto de las presentadas por la mayoría de las notaciones y lenguajes para modelado de procesos proviene de la descripción de roles o papeles jugados por los diferentes agentes que intervienen en el proceso y su interacción. Este tipo de enfoque comenzó con la aplicación de las ***Redes de Petri*** al modelado del proceso de coordinación en entornos de trabajo [Holt83] y fue proseguido en diferentes instituciones por el desarrollo de las ***Role Interaction Nets (RIN)*** [Sing92], técnica que modela la estructura de interacción de roles en un proyecto utilizando un lenguaje de representación basado en Redes de Petri.

Las RIN están diseñadas para facilitar la representación y ejecución de tareas estructuradas, presentando una gran potencia expresiva para representar roles, dependencias y elementos de proceso. No son, sin embargo, tan adecuadas a la hora de representar los artefactos (productos) del proceso.Cabe reseñar que, cuando se representa un proceso mediante RIN, emerge una nueva noción de equipo de trabajo más basado en dependencias entre roles que en jerarquías de responsabilidad. Cuando las interacciones entre roles son frecuentes, el conjunto de roles forma un grupo de trabajo de facto. Un plan de proyecto puede diseñarse como una descomposición de tareas, roles e interacciones. El plan de proyecto se establece cuando los roles se asignan a individuos concretos. Un individuo puede representar varios roles y un tipo de rol puede asignarse a más de un individuo.

Una vez instanciado un plan de proyecto, la RIN puede utili-

zarse como mecanismo para coordinar el encaminamiento de artefactos a través de la red de roles en interacción, así como mecanismo de trazado del nivel de progreso en función de la finalización de interaciones entre roles. En otras palabras, este formalismo parece adecuado como mecanismo de sopor-te a la gestión de actividades en entornos basados en proceso, aunque el uso práctico de este modelo para formalizar estructuras de roles complejas será dificil si la organización no ha podido establecer una descripción básica del proceso.

2.2. Modelado de procesos y metamodelo espiral

El ciclo de vida software es la representación de más alto nivel de abstracción del proceso de desarrollo software. En este sentido, queremos describir en esta sección la interrela-ción de las tareas de modelado, puesta en práctica y mejora del proceso de desarrollo software con las fases del ciclo de vida consideradas en los modelos de ciclo de vida clásicos.

El modelo de ciclo de vida en espiral constituye un modelo de síntesis o metamodelo que puede particularizarse en la mayor parte de los modelos de ciclo de vida propuestos en la literatura. De aquí que consideremos únicamente el caso del paradigma espiral para ilustrar su interrelación con el modelado del proceso software.En la literatura podemos encontrar experiencias de formalización del proceso de desarrollo en el marco del modelo en espiral. Merecen especial mención las realizadas por Boehm, el propio autor del modelo, y Belz (ver [Boeh89] y [Boeh90]), si bien se trata de experiencias preliminares utilizando lenguajes procedimentales (ADA), más enfocadas a la definición formalizada del proceso que a proporcionar un soporte para su ejecución automatizada.

Dichas experiencias, sin embargo, han resultado muy fructíferas en el sentido de que han conducido al autor del modelo a refinar y adaptar su definición inicial del modelo en espiral para albergar explícitamente el proceso de desarrollo.

La tesis principal de [Boeh89] es que la expresión formal del proceso de desarrollo (programa de proceso en el artículo original) será más efectiva si viene precedida por las fases habituales en todo proceso de desarrollo: requisitos, arquitec-tura, diseño, etc; y que el modelo en espiral representa un enfoque adecuado para representar y llevar a cabo la captura de requisitos, diseño de arquitectura y diseño detallado del proceso de desarrollo software. En nuestra opinión el modelo en espiral representa también un marco de referencia adecua-do para la ejecución y evolución del proceso de desarrollo.El resultado de la evaluación de las experiencias mencionadas condujo a Boehm a reformular el contenido del cuarto cuadrante (planificación de ciclo siguiente), desde su vaga propuesta inicial como segmento del ciclo de vida en que se realiza la planificación de etapas subsiguientes expresada en lenguaje natural en formularios más o menos normalizados en cada institución; hasta su nueva estructuración como una cadena de las actividades ya propuestas para los tres primeros cuadrantes: identificación de objetivos del proceso, análisis de alternativas, evaluación de riesgos, resolución de riesgos y evolución y validación del proceso de desarrollo. En resumen, la anterior cadena de actividades se realizaría dos veces en cada ciclo de desarrollo. Inicialmente para refi-nar los requisitos, arquitectura, diseño e implementación del producto a lo largo de los tres primeros cuadrantes; y luego para refinar los requisitos, arquitectura, diseño e implemen-tación del proceso de desarrollo en el cuarto cuadrante. La **Figura 1** presenta este refinamiento del modelo en espiral.

3.Integración de herramientas en sistemas CASE

La integración de herramientas ha sido reconocida a todos los niveles como un tema clave para el futuro de los entornos de desarrollo software. Esta permite la cooperación de muchas herramientas, en general procedentes de distintas fuentes, en un marco de integración uniforme y abierto.

La problemática de integración suele descomponerse en distinto aspectos. Por una parte la integración de datos tiene que ver con la compartición y el intercambio de datos entre herramientas. Este tipo de integración suele realizarse mediante modelos de datos que las distintas herramientas pueden interpretar. La integración de control, por otra parte, está más relacionada con la problemática de comunicación entre las herramientas. La integración a nivel de presentación es necesaria para proporcionar un aspecto uniforme y un modelo de interacción Único. Finalmente, cabe distinguir la integración de proceso, reflejo del grado en que las herramientas componentes de un entorno se coordinan con las actividades humanas para soportar el modelo de proceso pretendido. Expresado de forma sintética, los objetivos pretendidos por cada una de las perspectivas de integración mencionadas son los siguientes:

- ***Presentación:*** El objetivo de la integración de presentación es mejorar la eficiencia y efectividad de la interacción del usuario con el entorno optimizando la curva de aprendizaje.
- ***Datos:*** El objetivo de la integración de datos es asegurar que toda la información del entorno se gestiona como un todo consistente, sin entrar a considerar como se transforman o se utilizan partes de esta información.
- ***Control:*** Aquí el objetivo es permitir la combinación flexible de las diferentes funciones del entorno de acuerdo con las particularidades del proyecto y todo ello guiado por los procesos de desarrollo soportados por el entorno.
- ***Proceso:*** El objetivo de la integración con el proceso de desarrollo es asegurar que las herramientas interactuen eficientemente para soportar el proceso definido.

En el campo de los entornos de desarrollo software, estos aspectos se abordan de diferentes formas: integración total de datos [Thom89], arquitecturas de comunicación ba-

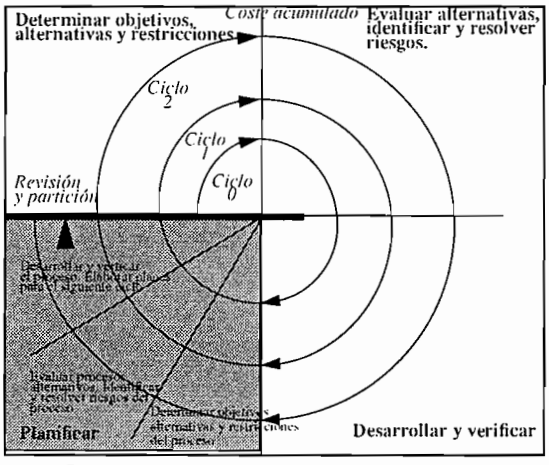


Figura 1: Refinamiento del modelo en espiral

sadas en mensajes [Reis88], interactuadores gráficos con *look and feel* normalizados (de facto) [OSF89], etc.
El objetivo de un entorno de ingeniería software es proporcionar soporte efectivo para un proceso de desarrollo software efectivo. Este soporte será más efectivo si el entorno es integrado, o sea si todos los componentes funcionan como partes de un todo Único consistente y coherente [Thom92]. En este sentido, merece la pena destacar los esfuerzos de es-tandarización de entornos de desarrollo y marcos de integra-ción realizados en el seno del ECMA a través de su Reference *Model for Frameworks of Software Engineering Environ-ments* [NE93], y en el NIST mediante su propuesta de estándar *Reference Model for Project Support Environments [*NC93]. Ambas instituciones han unificado sus propuestas en un modelo de referencia para arquitectura de entornos de desa-rrollo software que ha representado la arquitectura de refe-rencia para el entorno de desarrollo descrito en este trabajo.La propuesta NIST/ ECMA consta de un catálogo de descripciones de servicio que abarcan la funcionalidad de un entorno de desarrollo completo. Estas descripciones de servicio se agrupan en diferentes categorías segÚn criterios de nivel de abstracción, granularidad o funcionalidad. El más alto nivel de la clasificación agrupa a los diferentes servicios en dos categorías: servicios de usuario final y servicios de marco de integración, donde los servicios de usuario final hacen referencia a la funcionalidad de soporte al desarrollo propiamente dicho, es decir a la funcionalidad aportada por las herramientas de desarrollo comprendidas en el entorno. Por otra parte, los servicios de marco de integración hacen referencia a funcionalidades de infraestructura y soporte de integración y comunicación a las herramientas del entorno. Los servicios de marco de integración guardan una estrecha relación con las diferentes perspectivas de integración descritas más arriba en esta sección.La **figura 2** ilustra estos conceptos, mostrando los servicios de marco de integración relacionados con la integración de presentación, integración de proceso, integración de datos y plataforma de comunicación. Estos servicios constituyen la infraestructura en la que pueden insertarse las diferentes herramientas que implementan el subconjunto de servicios de usuario final que se considerasen de interés en cada caso.

3.1.Entornos basados en proceso

Como se ha mencionado en la sección 2, el modelado de procesos proporciona un nuevo enfoque para diseñar arquitecturas de entornos de desarrollo, denominados entornos o marcos de integración basados en proceso *(PCE/PCF, Process Centered Environments/Frameworks)*. En este tipo de entornos la especificación de los agentes que intervienen en el proceso de desarrollo, sus actividades y la forma en que éstas se coordinan entre sí y con las herramientas del entorno estaría definida en un modelo de proceso cuya repre-sentación computable constituiría el motor *(process-driver)* del entorno de desarrollo. El ***process-driver*** interpreta y ejecuta el modelo de proceso de acuerdo con su jerarquía de actividades, gestiona el orden de subtareas y las restricciones a satisfacer antes de que puedan llevarse a cabo, automatiza la inicialización y ejecución del modelo, acepta las entradas de los desarrolladores, modifica y propaga el estado de las tareas en ejecución y dispara el inicio de otras subtareas.
Hay dos características que distinguen a los PCF/PCE de otros tipos de entornos de desarrollo software:

- El énfasis explícito en los mecanismos relativos al proceso, su modelado y mecanismo de ejecución
- El énfasis en la comunicación e integración de las personas y sus acciones en lugar de en la comunicación e integración entre herramientas.

La **Figura 3** muestra de manera esquemática la arquitectura de un entorno basado en el proceso que incluye explícitamente la gestión de configuración y un gestor de presentación que proporciona integración visual.
En resumen, el entorno consta de un gestor de presentación que implementa la integración de presentación a la vez que proporciona diferentes perspectivas a los distintos usuarios. Las dos perspectivas principales son:

- Visión de desarrollo de procesos. El responsable de la implementación del modelo de proceso utilizará el entorno en su diseño y puesta en práctica tanto inicial como sus modificaciones sobre la marcha.
- Visión de desarrollo de aplicaciones. Los implementadores de aplicaciones utilizan el entorno en la realización de las aplicaciones finales siguiendo el modelo de desarrollo definido en la perspectiva anterior. Esta visión admite a su vez diferentes perspectivas mencionadas en la sección 2.

Por otra parte, el gestor de proceso coordina la interacción de los usuarios del entorno, a través del gestor de presentación, con el conjunto de herramientas en función del modelo de proceso ejecutado. Dicho modelo de proceso expresará una jerarquía de tareas, subtareas y acciones que tienen como atributos los objetos sobre los que actuan, los agentes que las realizan, los recursos requeridos etc. Todo ello defini-do con algun lenguaje gráfico o textual suficientemente forma-lizado y con la suficiente capacidad expresiva para soportar el análisis de dichos procesos y expresar convenientemente la ordenación e interdependencia de acciones y tareas. La **Figura 4** ofrece un descripción gráfica de estas ideas.

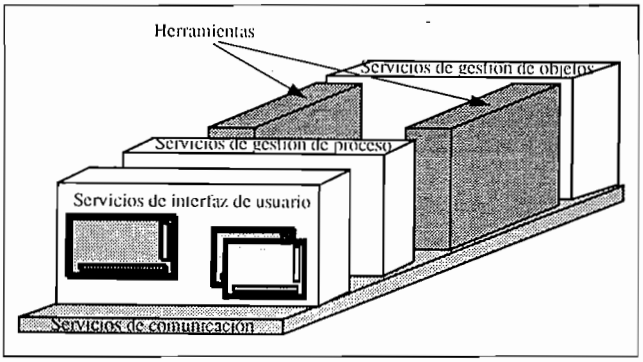


Figura 2: Ilustración de la arquitectura NIST-ECMA

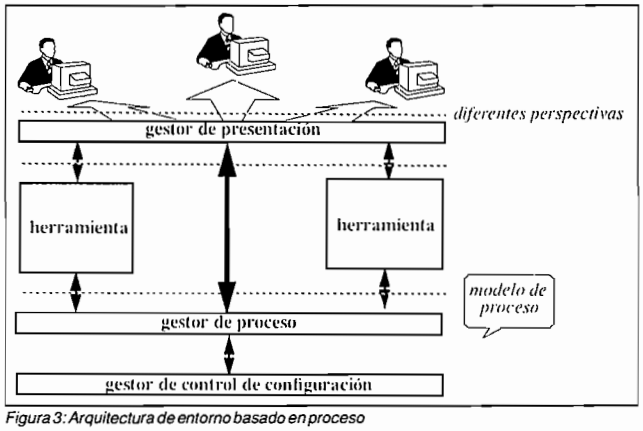


Figura 3: Arquitectura de entorno basado en proceso

Cada nivel en la descomposición jerárquica en subtareas y acciones debe especificar el orden parcial en la ejecución de dichas subtareas y acciones. En este sentido, puede haber varios tipos de relación de precedencia entre tareas: secuencial, concurrente, iteración, condicional, etc.
La Figura 3 no muestra un elemento imprescindible en todo entorno integrado: la infraestructura de comunicación entre gestores y herramientas que permite el intercambio de información entre ellos y proporciona, en alguna medida, la integración de control entre los componentes del entorno mediante algÚn mecanismo de encapsulación de las interfaces de las herramientas y de transferencia de información de control entre herramientas *(bus software)*. En este sentido, la conformidad respecto de estándares de facto en el terreno de la integración de herramientas es imprescindible para los PCE/PCF si han de interactuar con el cada vez más amplio mundo de las herramientas CASE y/o sobre plataformas de proceso distribuido. Algunas de las contribuciones más importantes en este sentido son la propuesta de plataforma distribuida CORBA [OX92], que parece converger con el estándar ODP de ISO [JTC194], PCTE [Wake93] como marco de integración a nivel de datos y el marco de integración a nivel de control ToolTalk [Juli92]. Tambien cabe reseñar la necesidad de integrar PCE/PCF con herramientas de gestión de configuración GCS). En la actualidad las herramientas para GCS suelen tener un modelo de proceso ***cableado*** (aunque en algunos casos ofrecen alguna capacidad de particularización a los gustos y necesidades del usuario) y ponen el acento en la gestión de producto más que en la gestión de proceso. De aquí la necesidad de que se produzca una convergencia entre los conceptos de automatización de proceso y gestión de configuración mediante la fusión de la gestión de datos y la gestión de proceso.

4. DELPHOS, entorno de desarrollo basado en proceso

Las secciones anteriores han presentado el contexto y estado del arte en relación con el modelado del proceso software, el modelo de referencia para soportar las diferentes perspec-tivas de integración en entornos de desarrollo y los entornos de desarrollo basados en proceso. A partir de este contexto se quiere presentar en esta sección la arquitectura y funcio-nalidad de un entorno de desarrollo integrado que está siendo desarrollado por el autor en el Departamento de Ingeniería de Sistemas Telemáticos de la Universidad Politécnica de Madrid (DIT/UPM). Se trata del entorno DELPHOS y sus características distintivas se relacionan a continuación:

- DELPHOS es un marco de integración basado en proceso con las siguientes características de servicio:
  - Integración de presentación basada en el estándar gráfico X-Window con el kit de desarrollo gráfico Tcl/Tk [Oust93].
  - Plataforma de comunicación basada en una implementación conforme al estándar OMG-CORBA para plataformas de objetos distribuidos [OX92].
  - Integración de proceso basada en una notación gráfico-textual para descripción de proyectos, actividades, agentes y variables de proceso y una notación de ejecución basada en Redes de Petri temporizadas de alto nivel [Feld93].
  - El soporte a la integración de proceso está fuertemente interrelacionado con la gestión de configuración, cambios y versiones de los productos del desarrollo a través del gestor de configuración Aegis [Mill93].
- El campo de aplicación del entorno es la especificación, diseño e implementación de servicios avanzados de telecomunicación mediante elementos de servicio que interoperan a través de una plataforma de objetos distribuidos del tipo OMG-CORBA.
- El tipo de metodología a la que se quiere dar soporte tiene como componentes principales el uso del paradigma de orientación a objetos, en particular OMT [Rumb91] y el uso de Técnicas de Descripción Formal como LOTOS [ISO89] y MSC [IT93] para especificación de comportamiento, casos de uso y propósitos de prueba.

El entorno DELPHOS se encuentra actualmente en fase de prototipo. Si embargo, su arquitectura y las notaciones de modelado se encuentran suficiente perfiladas. En las siguientes subsecciones se ofrece una breve descripción de las mismas con un pequeño ejemplo ilustrativo.

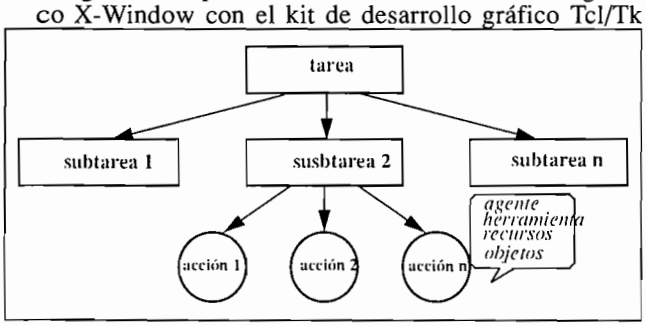


Figura 4: Estructura general de un modelo de proceso

4.1. Notaciones de modelado

Las notaciones de modelado del entorno DELPHOS se basan en la utilización de dos notaciones: una **notación de usuario** y otra de **ejecución**. La **notación de usuario** se basa fuertemente en formalismos gráficos que proporciona al usuario un medio de expresión del proceso de desarrollo de aplicaciones sencillo de manejar, ofreciendo una visión intuitiva del modelo de desarrollo representado. La notación de usuario que aquí se propone es la expresión gráfica de un **proyecto** de desarrollo software a través de las actividades que lo componen. Dichas actividades se caracterizan por una serie de atributos como: agentes y roles que participan en el desarrollo de la actividad, recursos necesarios, requisitos temporales, productos de entrada, productos de salida, entorno de herramientas, etc. Por otra parte, las actividades en que se descompone un proyecto se relacionan entre sí a través de operadores como: precedencia, concurrencia, iteración, habilitación, sincronización, etc.
Por otra parte, también se ofrece una **notación de ejecución** que permite la expresión precisa y computable de dichos modelos de proceso de forma que resulte adecuada para materializar la integración de proceso en el entorno de desarrollo. Esta podrá derivarse de forma automática desde la expresión de procesos mediante la notación de usuario, por utilización directa o mediante la utilización de bibliotecas de modelos de proceso. Se ha optado por el uso de Redes de Petri de Alto Nivel temporizadas *(HLTPN: High Level Timed Petri Nets)* con alguna pequeña modificación sintáctica para su mejor adecuación al modelado de procesos.
La idea general es proponer una máquina de estados en forma de HLTPN para cada conector de actividades y una expresión general de actividad que puede utilizarse como plantilla genéri-ca que el usuario puede utilizar tal cual o bien

modificar para adaptarla a sus necesidades concretas y que lleva implícita la definición genérica de la interacción con el entorno de Gestión de Configuración Software (GCS) que permita gestio-nar la *baseline* (repositorio de ficheros fuente) del proyecto.

En la **figura 5** se representa la notación básica de proyecto a través de los operadores de relación entre actividades más habituales. En principio, se supone una estructura de proyecto aplanada. Es decir, los operadores no representan descomposición jerárquica de actividades, sino relaciones temporales de precedencia y sincronización entre actividades, aunque mediante el uso de una actividad sin comportamiento propiamente dicho se puede representar la descomposición jerárquica de actividades. El operador de iteración puede representarse mediante la secuencia de una actividad consigo misma o con otra precedente. De forma análoga se puede pensar en otros operadores que probablemente puedan representarse en términos de los operadores básicos. La notación que se muestra en la figura se utilizará para elaborar la ***perspectiva funcional*** de los modelos de proceso. Esta notación se completará más adelante con la notación OMT para proporcionar la ***perspectivas informacional y organizacional*** sobre los modelos definidos.

En principio el objetivo es encontrar una representación Única y general para las actividades y los operadores en términos de HLTPN, incluso parece viable obtener una Única representación genérica para cada uno de los diferentes operadores. Esta notación para la descripción de actividades basada en HLTPN permite elaborar la ***perspectiva de comportamiento*** de los procesos de desarrollo.Una actividad puede representarse en forma de HLTPN por una serie de lugares (o estados) y transiciones que representan cambios de estado interno de la actividad o habilitación de la actividad subsecuente a través de lo conectores de actividad anteriormente mencionados. En esta notación las transiciones de la HLTPN se descomponen en dos *subtransiciones*. La primera viene a representar la satisfacción de las precondiciones de disparo, que en nuestro caso son condiciones del tipo: ocurrencia de evento externo, espera de toma de decisión por parte de un agente, espera de valor de retorno de un procedimiento y, por supuesto, la satisfacción de una expresión temporal o lógica sobre las variables del proceso. En cuanto a la segunda *subtransición*, representa la ejecución de una acción asociada a la transición como, por ejemplo: envio de notificaciones o contextos de trabajo, ejecución de una función escrita en lenguaje de shell o procedimental, llamada a un subproceso, etc. En la **figura 6** se representa una actividad simple que inicialmente se encuentra inactiva (IDLE) hasta que se verifican las condiciones de inicialización de la actividad (por ejemplo la aceptación por parte del agente que debe llevar a cabo la actividad), en cuyo caso se realizan las acciones asociadas a la transición (en este caso podría ser la inicialización de atributos de actividad y registro en el histórico del proyecto). Esta transición lleva a la actividad al estado READY. La transición posible desde este estado no tiene precondiciones (siempre *true*) y las acciones de la transición podrían ser la invocación de un procedimiento (escrito en cualquier lenguaje de programación o de shell) que represente el detalle de la ejecución de la actividad: incluir la tarea en la lista de tareas pendientes del agente asociado, proporcionar información de contexto para la particularización del entorno de herramientas, realizar la petición de los ficheros de partida al GCS, etc. Esta Última transición situa la actividad en estado IN-PROGRESS hasta que se produce el retorno del procedimiento invocado, en cuyo caso se pueden realizar las acciones apropiadas aso-ciadas al GCS y el registro en el histórico de proyecto. Una vez finalizadas estas acciones, la actividad pasa a estado END y se considera finalizada. Por otra parte, la semántica de los ope-radores de relación de actividades también se representa en términos de HLTPN. Como ejemplo se muestran los siguientes:

- AND: Las tareas de entrada al conector deben finalizar todas para que se considere habilitada la actividad de salida.
- OR: Una cualquiera de las actividades de entrada debe finalizar para que se considere habilitada la actividad de salida.

En la **figura 7** se muestra un diagrama de estados general para una actividad más realista. Una vez creada, la actividad pasa al estado READY donde espera a ser habilitada para pasar de forma iterativa por una serie de peticiones de cambio que representan el desarrollo, prueba, revisión, integración y mantenimiento de parte o la totalidad del sistema en desarrollo. Una tarea activa (IN_PROGRESS) puede suspen-derse y reanurse en cualquier momento. Cuando un cambio se termina, prueba e integra pasa al estado COMPLETED, desde donde puede modificarse de nuevo después(*change*) o considerarse finalizada definitivamente (DONE). Queda aún abierta la cuestión de si la actividad habilita al conector de salida desde el estado COMPLETED o DONE.El estado IN_PROGRESS encapsula las tareas de desarrollo, prueba, revisión e integración de los productos en desarrollo. La parte derecha de la **figura 7** refina dicho estado en función del modelo de petición de cambios habitual en las herramien-tas de gestión de configuración. Este refinamiento es particu-larizable en función del GCS utilizado. En el mostrado en la figura, la tarea comienza en espera de comienzo del esfuerzo de desarrollo, para pasar por los estados de desarrollo, revisión, espera de integración e integración en la *baseline*.

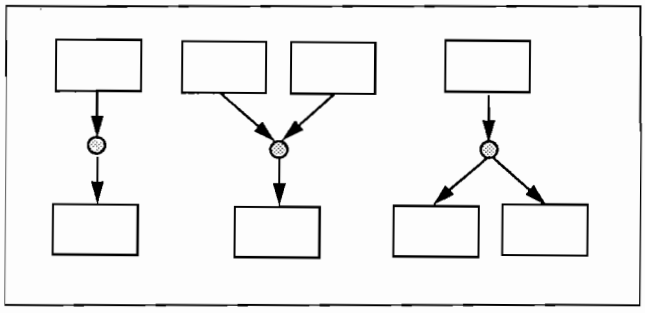


Figura 5: Representación básica de modelos de proyecto

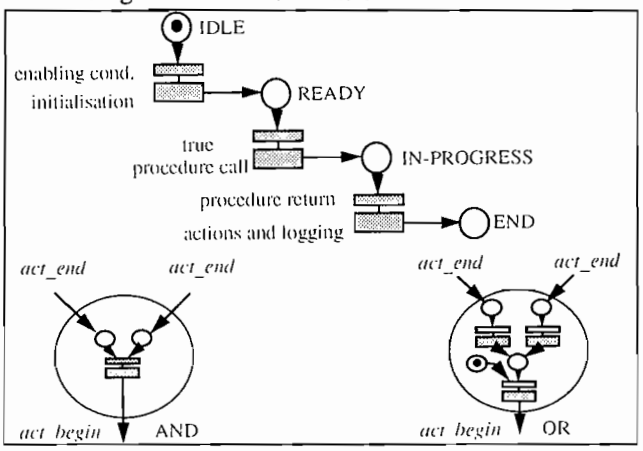


Figura 6: Representación simplificada de actividad y conectores básicos

4.2. Arquitectura del entorno

El objetivo general del entorno DELPHOS es proporcionar un marco conceptual y un conjunto de herramientas para la definición, gestión y ejecución de los elementos de proceso de desarrollo que forman parte de tareas, actividades y proyectos de desarrollo software. En este sentido, el entorno se puede considerar compuesto por dos subentornos de funcionalidades claramente diferentes, aunque interrelacionadas:

- *Subentorno de desarrollo de procesos*. Es el entorno de soporte a la definición, instanciación, ejecución y gestión de modelos de proceso de desarrollo. Consta de un conjunto de notaciones gráfico-textuales junto con sus herramientas asociadas para la definición formalizada y ejecución guiada de un proyecto de desarrollo software. En general, el entorno ha de permitir la descripción de un proyecto software a distintos niveles de abstracción: ciclo de vida, estructura jerárquica de actividades, atributos de actividad (entradas, salidas, roles...), descripción procedimental de la actividad, etc.
- *Subentorno de usuario*. Es el entorno que proporciona soporte al usuario en la tarea de desarrollo de los productos del proyecto: documentos, diseños, programas, etc. Este subentorno se instancia para cada usuario en función del proceso ejecutable asociado a la actividad en desarrollo.

4.2.1. Subentorno de desarrollo de procesos

El subentorno para desarrollo de procesos es el componente de DELPHOS que da soporte al desarrollo de procesos soft-ware con diferentes niveles de granularidad. Al más alto nivel de abstracción, este subentorno permite la descripción de proyectos de desarrollo a través de la jerarquía de actividades que componen el proyecto. El refinamiento de esta repre-sentación prosigue con la especificación de cada actividad del proyecto mediante cumplimentación de los atributos de actividad como entradas, salidas, roles, etc y, finalmente, la descripción formalizada de los pasos de proceso necesarios para llevar a cabo dicha actividad. Consta de estos editores:

- *Editor de proyectos*. Es la herramienta principal del subentorno de desarrollo de procesos y a través de la cual se invocan el resto de componentes del subentorno. Su función principal es la definición y ejecución de descripciones de proceso de alto nivel utilizando la notación gráfica de usuario propuesta más arriba. Es decir, proyectos de desarrollo completos (perspectiva funcional). Su funcionalidad incluye la creación de la *baseline* del proyecto y el registro del proyecto y sus atributos en la herramienta de gestión de configuración asociada.
- *Editor de actividades*. Por invocación desde el editor de proyectos y sobre cada actividad, este editor permite la configuración de una actividad en cuanto a particularización de sus atributos: entradas, salidas, roles, etc y a su enlace a un elemento de proceso que especifique la actividad a nivel de comportamiento.
- *Editor de procesos*. El editor de actividades junto con los atributos mencionados proporcionará, a través del editor de procesos, descripciones detalladas de los pasos de proceso que componen la actividad (perspectiva de comportamiento). Las transiciones de la red representarán aquí las precondiciones, procedimientos y postcondiciones, asociados a un paso de proceso. La evaluación de condiciones y disparo de transiciones se relacionará con las condiciones de habilitación propias de la notación: tokens habilitando la transición y condiciones lógico-temporales evaluando a cierto; y a la ejecución de código procedimental asociado a dichas condiciones y procedimientos.

El resultado de los procesos especificados a través de los editores mencionados queda recogido en una serie de ficheros de datos, *shell* y HLTPN. Estos Últimos se compilan a código C++ mediante un compilador desarrollado previamente en DIT/UPM. Dicho código, junto con las especificación de interfaces (IDL), constituyen la base para generar los diferentes servidores que componen el entorno en forma de aplicación de objetos distribuidos sobre plataforma CORBA.

4.2.2. Subentorno de usuario final

La funcionalidad de este subentorno es la de un entorno de desarrollo basado en proceso. Sus usuarios no tienen necesariamente que conocer los detalles del proceso que guía su entorno de trabajo, aunque pueden contribuir al desarrollo o enriquecimiento de los procesos de desarrollo utilizados. Además podrá utilizar el entorno como gestor de tareas personales y medio de comunicación con los componentes de su equipo de trabajo. Otra funcionalidad del entorno es la de encapsular y automatizar el control de configuración, versiones y cambios de proyecto.

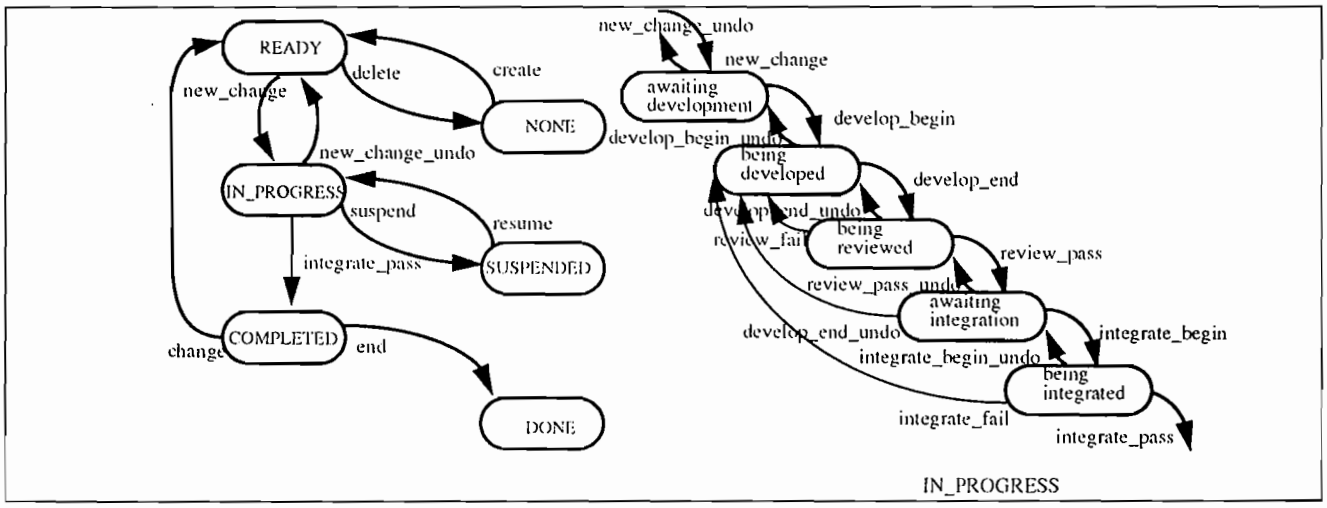


Figura 7: Diagrama de estados del proceso actividad

En general, cada tarea de usuario tiene asociado un entorno integrado de herramientas como contexto de trabajo para llevar a cabo la tarea involucrada. El modelo de integración del entorno está basado en el modelo propuesto por las instituciones de estandarización NIST-ECMA. Los componentes básicos del subentorno de usuario son: la agenda de usuario, el contexto de trabajo y el entorno de herramientas de desarrollo.

- *Agenda de usuario*. Es el elemento central del entorno de usuario final y centraliza la gestión de las tareas personales del usuario, sean éstas tareas personales aisladas (puntuales o periódicas) o actividades de un proceso más complejo (actividades de desarrollo). Las tareas se representan mediante iconos que permiten enlazar con los contextos de trabajo asociados a dichas tareas.
- *Contexto de trabajo*. Es el elemento del entorno asociado con una tarea y proporciona los elementos de trabajo necesarios para llevar ésta a cabo: petición de cambio, atributos de la actividad, botones de decisión. Estos elementos serán característicos del rol del usuario en la tarea: desarrollador, revisor, integrador, administrador, etc
- *Entorno de herramientas (propiamente dicho)*. Se instancia para cada actividad en progreso en función del entorno de herramientas definido por defecto para el proyecto y el rol del usuario que lo invoca.

4.2.3 Integración de componentes

En las secciones precedentes se ha hecho una descripción de los componentes básicos de DELPHOS para la definición de procesos a diferentes niveles de abstracción: proyecto, actividad, procedimiento. También se han descrito los elementos que constituyen la interfaz de usuario final del entorno: agenda, contexto de trabajo, entorno de herramientas. Esta sección ofrece una visión de alto nivel de la arquitectura de soporte a la ejecución de dichos elementos, así como su estructura de integración e intercomunicación. La **figura 8** ofrece una aproximación a la arquitectura que consta de:

- ***Gestor de proyecto***. Es el soporte de ejecución de la definición de proyecto realizada a través de editor de proyectos. Lanza la ejecución (remota) de actividades y conoce el estado de ejecución de las diferentes actividades y el conjunto de agentes y artefactos que constituyen el proyecto. También se encarga de recolectar métricas a nivel de proyecto (e.g. nivel de progreso global y por actividades).
- ***Gestor de actividad***. Es el soporte de ejecución de cada actividad. Instancia cada actividad y sus parámetros tales como artefactos de entradas, roles, proyecto, baseline, agente inicial, etc.
- ***Repositorio***. Representa el almacén físico de los productos del proceso y del propio proceso. Está constituido por la baseline de la herramienta de gestión de configuración junto con los diferentes componentes del modelo de proyecto, sus datos de configuración y su estado persistente. Aunque se representa centralizada, la información del repositorio puede estar distribuida.
- ***GCS y proxi***. La herramienta de gestión de configuración software (GCS) y un representante que permite la invocación remota de sus comandos de operación. Se trata de un servidor CORBA que actua de intermediario. En principio, esta configuración se simplifica si las máquinas involucradas comparten el sistema de ficheros.
- ***Plataforma de comunicación***. Para el intercambio de información de control entre gestores del proceso la plataforma es actualmente una implementación de la plataforma CORBA.

La **figura 9** muestra la estructura de integración del entorno de herramientas asociado a una actividad.

5. Conclusiones y trabajo futuro

En este artículo hemos intentado sintetizar los conceptos básicos y el estado del arte de una disciplina denominada genéricamente modelado de procesos. Nuestro interés por dicha disciplina se debe a su aplicación en ingeniería del software, donde abre la posibilidad de multitud de aplicaciones y líneas de investigación que se han mencionado a lo largo del artículo. En concreto nuestro interés recae en la aplicación del modelado formal del proceso de desarrollo software como elemento integrador de proceso en sistema CASE. En este tipo de entornos la especificación de los agentes que intervienen en el proceso de desarrollo, sus actividades y la forma en que éstas se coordinan entre sí y con las herramientas del entorno estaría definida en un modelo de proceso cuya representación computable constituiría el motor del entorno de desarrollo. Esta perspectiva de integración esta ganando relevancia como factor de integración en entornos de desarrollo. Prueba de ello es su inclusión como servicio de gestión de proceso en el Modelo de Referencia para Entornos de Soporte a Proyectos elaborado por las instituciones de estandarización NIST y ECMA. A partir de este contexto se ha presentado la arquitectura y funcionalidad de un entorno de desarrollo integrado que está siendo desarrollado por el autor en el DIT/UPM. Se trata del entorno DELPHOS, un marco de integración basado en proceso con las siguientes características de servicio:

- Integración de presentación basada en el estándar gráfico X-Window a través del kit de desarrollo gráfico Tcl/Tk.
- Plataforma de comunicación basada en una implementación conforme al estándar OMG-CORBA para plataformas de objetos distribuidos.
- Integración de proceso basada en una notación gráfico-textual para descripción de proyectos, actividades, agentes y variables de proceso y una notación de ejecución basada en Redes de Petri temporizadas de alto nivel.
- Soporte a la integración de proceso interrelacionado con la gestión de configuración, cambios y versiones de los productos del desarrollo.

El campo de aplicación del entorno es la especificación, diseño e implementación de servicios avanzados de telecomunicación mediante elementos de servicio que interoperan a través de una plataforma de objetos distribuidos del tipo OMG-CORBA. El tipo de metodología a la que se quiere dar soporte tiene como componentes principales el uso del paradigma de orientación a objetos y el

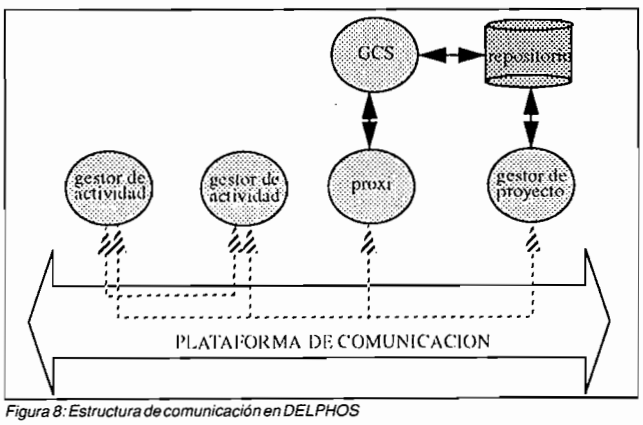


Figura 8: Estructura de comunicación en DELPHOS

uso de Técnicas de Descripción Formal para especificación de comportamiento, casos de uso y propósitos de prueba.

En la actualidad se dispone de un primer prototipo del entorno DELPHOS con la infraestructura de integración y se trabaja en la incorporación al entorno del conjunto de herramientas de desarrollo tales como: editor de texto, editor/generador de código basado en OMT, entorno de desarrollo LOTOS, compilador IDL, etc.

Para el futuro se prevé la ampliación del entorno con objeto de incluir capacidad de análisis dinámico *off line* de descripciones de proceso, animación gráfica de descripciones de proceso, soporte a la reutilización de elementos de proceso a través de una biblioteca de procesos reutilizables, soporte al trabajo cooperativo e incorporación de la recolección automática de métricas de proceso.

6. Referencias

[Band93] S.C Bandinelli, A. Fuggetta, and C. Ghezzi. *Software process model evolution in the spade environment.* IEEE Transactions on Software Engineering, 19 (12):1128ñ1144, December 1993.

[Boeh89] B. Boehm and F. Belz. *Applying Process Programming to the Spiral Model.* In Tully Tull89. ACM SIGSOFT Software Engineering Notes, Vol. 44, No. 4, June 1989.

[Boeh90] B. Boehm and F. Belz. *Experiences with the Spiral Model as a Process Model Generator.* In Dewayne E. Perry, editor, *Proceedings of the 5th International Software Process Workshop*, Kennebunkport, Maine (USA), 1990. IEEE/ACM, IEEE Computer Society Press.

[Curt92] B. Curtis, M. I. Kellner, and J Over. *Process modeling.* Communications of the ACM, 35(9):75ñ90, September 1992.

[Feld93] M. Felder, C. Ghezzi, and M. Pezzè. *High-level timed Petri nets as a kernel for executable specification.* Real-Time Systems, 5(2/3):235ñ249, Mayo 1993.

[Hare90] D. et al Harel. *Statemate: A working environment for the development of complex reactive systems.* IEEE Transactions on Software Engineering, 16(4):403ñ412, April 1990.

[Holt83] A. Holt, H.R. Ramsey, and J. Grimes. *System technology as a basis for a programming enviroment.* ITT Electr. Commun, 57(4), April 1983.

[Huff89] K.E. Huff and V.R. Lesser. *A plan-based intelligent assistant that supports the software development process.* In Conference Proceedings. Software Eng.Not., 1989. 13, 5, pp. 202ñ217.

[IEE91] IEEE Computer Society. *First International Conference on the Software Process*, Washinton D.C., 1991. IEEE Computer Society. pp. 202ñ217.

[ISO89] ISO. *Information Processing Systems - Open Systems Interconnection ñ LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* IS-8807. International Standards Organization, 1989. [published 15/feb/1989].

[IT93] ITU-T. *Message Sequence Charts.* Q. ITU-T, 1993. [].

[JTC194] ISO/IEC JTC1/SC21/WG7. *Basic Reference Model of Open Distributed Processing.* ISO/IEC 10746-1, ITU-T X.901-4. ISO/ITU-T, 1994. [].

[Juli92] A. Julienne and B. Holtz. *Designing and Writing a ToolTalk Procedural Protocol.* Working Paper, SunSoft, July 1992.

[Kell89] M.I. Kellner. *Representation Formalisms for Software Process Modelling.* In Tully Tull89. ACM SIGSOFT Software Engineering Notes, Vol. 44, No. 4, June 1989.

[Mill93] Peter Miller. *Aegis: A Project Change Supervisor.* Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA, August 1993.

[NC93] NIST-CMU/SEI. *Reference Model for Project Support Environments.* CMU/SEI 93-TR-23, NIST SP 500-213. NIST-CMU/SEI, 1993. [].

[NE93] NIST-ECMA. *Reference Model for Frameworks of Software Engineering Environments.* ECMA TR/55, NIST SP 500-211. NIST-ECMA, 1993. [].

[OSF89] OSF. *OSF/Motif Programmers Guide.* Working Paper Revision 1.0, Open Software Foundation Inc., November 1989.

[Oste87] L. Osterweil. *Software Processes are Software Too.* In 9th International Conference on Software Engineering, Monterey, California (USA), 1987. IEEE, IEEE Computer Society Press.

[Oust93] J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, 1993. [ISBN: 0-201-63337-X].

[OX92] OMG-X/OPEN. *Common Object Request Broker: Architecture and Specification.* Working Paper OMG TC Document 91.12.1, Object Management Group, Framingham, MA (USA), September 1992.

[Reis88] S. Reiss. *Integration Mechanisms in the Field Environment.* Working Paper Technical Report No. CS-88-18, Computer Science Department Brown University, October 1988.

[Rumb91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall International Editions, 1991.

[Sa92] J. Sa and B. Warboys. *Integrating a Formal Specification Method and PML. A Case Study.* In J.C. Derniame, editor, *Software Process Technology*, Trondheim (Norway), 1992. Springer-Verlag. Lecture Notes in Computer Science 635.

[Sing92] B. Singh and G.L. Rein. *Role Interaction Nets (RINs): A Process Description Formalism.* Technical Report CT-083-92, Microelectronics and Computer Technology Corp., Austin, Tex., April 1992.

[Sutt90] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. *Language constructs for managing change in process-centered environments.* In *Proceedings of ACM SIGSOFT t90: Fourth Symposium on Software Development Environments (SDE4)*, pages 206-217, Irvine, California, December 1990. Published as ACM SIGSOFT Software Engineering Notes, vol. 15, no. 6, December 1990.

[Suzu91] M. Suzuki and T. Katayama. *Metaoperations in the process model HFSP for the dynamics and flexibility of software processes.* In *Conference Proceedings* IEE91. pp. 202-217.

[Thom89] I. Thomas. *Pcte interfaces: Supporting tools in software-engineering environments.* IEEE Software, November 1989.

[Thom92] I. Thomas and B. Nejmeh. *Definitions of tool integration for environments. IEEE Software*, pages 29-35, March 1992.

[Tull89] Collin Tully, editor. *4th International Software Process Workshop*, Moretonhampstead (UK), 1989. IEEE/ACM, ACM Press. ACM SIGSOFT Software Engineering Notes, Vol. 44, No. 4, June 1989.

[Wake93] L. Wakeman and J. Jowett. *PCTE. The standard for open repositories.* Prentice Hall, 1993.

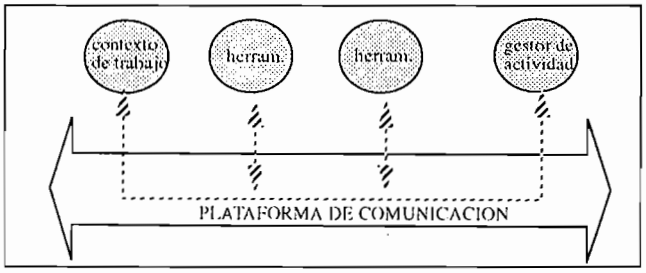


Figura 9: Integración de herramientas en DELPHOS

Ingeniería del Software

Javier Tuya, José R. de Diego, José A. Corrales
Área de Lenguajes y Sistemas Informáticos
Universidad de Oviedo
E-Mail: tuya@etsiig.uniovi.es

Verificación de Sistemas en Tiempo Real utilizando Comprobadores de Modelos

Resumen: *en este artículo se muestra cómo se pueden utilizar comprobadores de modelos para realizar la verificación formal de propiedades de vivacidad y seguridad en especificaciones gráficas obtenidas con métodos SA/RT. El comportamiento reactivo del sistema se modela de forma operacional mediante diagramas de transición de estados. Esta especificación es acompañada de las propiedades que ha de cumplir, escritas de forma declarativa en lógica temporal CTL. Se utiliza entonces un comprobador de modelos (SMV) para verificar el cumplimiento de las propiedades deseadas en el modelo operacional.*

1. Introducción

Los sistemas en tiempo real son a menudo sistemas críticos, de cuyo correcto funcionamiento puede depender la vida de las personas. Existen muchos ejemplos recientes de graves fallos como el del sistema de radioterapia Therac-25, que entre 1985 y 1987 se vio involucrado en seis accidentes, causando la muerte a muchos pacientes. Las causas de estos fallos fueron analizadas [11] y se pudo comprobar que en el fondo radicaban en una insuficiente utilización de principios generales de Ingeniería del Software (diseño complicado, insuficiencia de pruebas, etc.).

Los métodos estructurados para análisis y diseño de sistemas en tiempo real (SA/RT) [23], [10] y otros inspirados en éstos tales como Statecharts [9] son una extensión de los métodos estructurados 'clásicos' que utilizan Diagramas de Flujo de Datos (DFD) a los que se añade, entre otros aspectos, la posibilidad de expresar el comportamiento reactivo que caracteriza la mayor parte de este tipo de sistemas. Estos métodos son fáciles de utilizar y bien aceptados en la industria.

Puesto que la descripción del comportamiento reactivo del sistema se realiza mediante la definición de los estados del sistema y las condiciones ante las cuales éste cambia de estado (utilizando Diagramas de Transición de Estados, STD), este comportamiento puede ser complejo y muy difícil de verificar utilizando técnicas de inspección. Las herramientas CASE disponibles se suelen limitar exclusivamente a algunas comprobaciones sintácticas de corrección y completitud.

Por otra parte, los métodos formales [15] tienen una fuerte base matemática y permiten un enfoque más riguroso en el desarrollo de este tipo de sistemas. Con ellos se obtienen especificaciones precisas, carentes de redundancias, y se dispone de la posibilidad de aplicar técnicas de verificación formal. Este tipo de métodos son mucho más utilizados en el ámbito académico que en la industria.

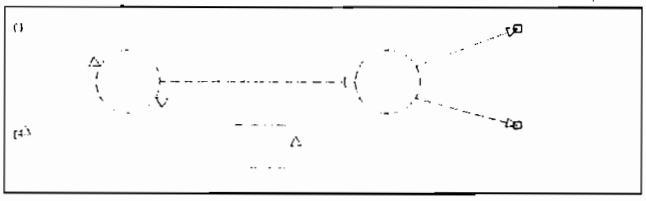


Figura 1: Transformaciones de Control

El estudio de la utilización de métodos formales en el ámbito industrial [6] ha detectado, entre otros, la evidente necesidad de unificar ambos enfoques, utilizando notaciones más adecuadas para el ingeniero medio (fundamentalmente gráficas). Por ello, en este artículo se propone la utilización de un formalismo gráfico basado en los métodos SA/RT junto con una lógica temporal, con el objeto de poder verificar determinadas propiedades que se han de cumplir en el sistema. Estas propiedades pueden ser de seguridad (algo que no puede suceder nunca) o de vivacidad (algo que debe suceder).

1.1. Métodos SA/RT

Una especificación SA/RT consta de una serie de diagramas estructurados en los cuales, además de procesos que transforman datos, existen procesos que transforman control (**Figura 1**). Cada proceso (transformación) de control tiene como entrada y salida flujos de control que representan eventos: sucesos que ocurren en un instante de tiempo. Estos eventos permiten al sistema comunicarse con el exterior (manejar dispositivos y recibir señales de dispositivos), o con otros procesos.

El comportamiento de cada proceso se especifica mediante diagramas de transición de estados (**Figura 2**). Cada proceso siempre se hallará en un estado, aunque el estado puede cambiar cuando se ejecuta una transición. Una transición (representada por una flecha) se produce cuando aparece determinada condición. Como consecuencia de la ejecución de la transición, el proceso correspondiente realiza el cambio de estado y se emite una acción. Tanto condición como acción están formadas por flujos de control.

Según el método de Ward/Mellor, el desarrollo del sistema se divide en dos fases que coinciden con las etapas típicas de análisis y diseño: En primer lugar se crea el modelo esencial (modelo lógico del sistema, sin tener en cuenta ningún detalle dependiente del entorno tecnológico). De este deriva el

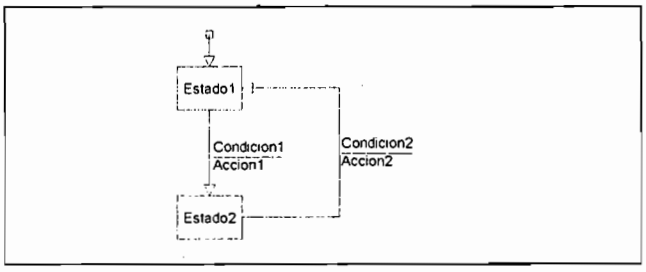


Figura 2: Diagramas de Transición de Estados

modelo de implementación (modelo físico). Siempre se ha de intentar seguir el principio de la minimización de la distorsión del modelo esencial, que consiste en que el modelo físico esté derivado a partir del lógico con la mínima modificación de éste.

1.2. Verificación Formal

En general, las actividades de verificación y validación de un sistema tienen como objetivo conseguir un sistema con el menor número de defectos posibles. Típicamente incluyen inspecciones y pruebas. Mientras que en el proceso de prueba del software se asume que siempre existirán defectos latentes, en la verificación formal se busca la garantía de la ausencia de defectos. Puede intuirse por tanto, la elevada complejidad del problema. La verificación formal puede realizarse según dos enfoques diferentes:

Prueba de teoremas: consiste en un proceso deductivo, que necesita un gran conocimiento matemático por parte del usuario.
Comprobación de modelos: consiste en la ejecución de algún programa (el comprobador de modelos) que determina de forma automática si la propiedad que se desea verificar es cierta o no. Es mucho más sencillo de utilizar que la prueba de teoremas.

1.3. Especificación y Verificación

En este trabajo se ha adoptado lo que se denomina enfoque 'dual' (dual language approach) para especificación y verificación, en el cual se parte de una especificación operacional, derivada de un método SA/RT como Ward/Mellor que representa de forma gráfica el comportamiento del sistema. Esta especificación es enriquecida con una serie de propiedades escritas de forma declarativa en lógica temporal CTL [5]. El proceso de verificación consiste en comprobar que dichas propiedades son ciertas en el modelo anterior, lo cual se realiza con un comprobador de modelos denominado SMV [14] basado en árboles binarios de decisión [3].

Para poder realizar una verificación formal de una especificación es totalmente necesario que ésta esté expresada formalmente, de forma matemática y sin ambigüedades. En el caso de los métodos SA/RT, que son considerados lenguajes gráficos semi-formales, es preciso en primer lugar describirlo formalmente, lo que se realiza en la Sección 2. El modelo gráfico que representa la especificación operacional ha de ser traducido al lenguaje utilizado por el comprobador de modelos, lo cual se trata brevemente en la Sección 3.. Tras ello, es posible realizar la verificación de las propiedades del modelo (Sección 4.). Se proporciona un método para reducir el tamaño del modelo como paso previo a la verificación, ilustrando todo ello con un ejemplo.

A lo largo de todo el artículo se ha intentado proporcionar una descripción lo más simplificada posible, huyendo de complicaciones en la notación matemática. Todo lo tratado en este artículo se puede encontrar con más detalle en [21] y [22].

2. Sintaxis y Semántica del Modelo Gráfico

Existen diversas formas de definir la semántica de los modelos SA/RT, siendo una de las más utilizadas las redes de Petri [16], [7], que aunque resultan complicadas, permiten aprovecharse de su ejecutabilidad y de sus posibilidades de verificación de propiedades [8]. Puesto que nuestro objetivo es verificar propiedades relativas al comportamiento reactivo del sistema, que en SA/RT se expresan mediante diagramas de transición de estados, el modelo se puede definir de una forma más sencilla como un sistema de transiciones en el cual los componentes del modelo gráfico (flujos, estados, procesos) son definidos matemáticamente mediante variables.

Este sistema de transiciones se representa siguiendo la notación de Manna y Pnuelli [13] como $\Gamma, \Sigma, T, \Theta_0$, cuyos componentes son:

1. Variables de Estado Global: $\Gamma=\Pi\cup F$. Representan cada uno de los elementos del sistema (procesos, flujos). Cada variable $\pi\in\Pi$ (denominada variable de estado local) define los posibles estados de cada proceso. Cada Proceso P tendrá asociado una variable de estado local π cuyos valores se corresponden con los de la representación gráfica.

Cada variable de $f\in F$ (denominada flujo) representa un flujo de control que comunica procesos entre sí o con el exterior. Se distingue entre flujos de evento (F^e) y flujos de variable (F^v). Los primeros representan comunicación por medio de eventos y los segundos por medio de variables compartidas (similar a la utilización de almacenes de control en los métodos SA/RT). Los flujos de control podrán ser además asíncronos (F^a) y síncronos (F^s). La diferencia se tratará más adelante.

2. Estados Globales: Σ. Definen cada uno de los posibles estados del sistema en su totalidad (serán las posibles combinaciones de valores de las variables de estado global Γ). Se define estado local s de un proceso cuya variable de estado local es π como el conjunto de estados globales tales que $\pi=s$.

3. Transiciones: T. Representan los cambios de estado. Se utiliza la notación $\tau: s \xrightarrow{c/a} s'$ para indicar que la transición τ causa que un proceso cambie del estado local s al s' cuando aparece una condición c. Al ejecutarse la transición se produce una acción a. Tanto c como a están formadas por flujos. Formalmente, la transición se define utilizando la relación de transición, que en forma general se expresa como:

$$\rho_\tau: C_\tau(\Gamma) \wedge A_\tau(\Gamma') \qquad (1)$$

El primer término representa los valores de todas las variables del sistema de transiciones antes de la ejecución de la transición y el segundo los valores después de la transición (denotados con prima).

4. Condición inicial: Θ_0. Es el estado del sistema en el instante del comienzo de la ejecución.

2.1. Transiciones Asíncronas

Cuando un proceso ejecuta una transición asíncrona, el proceso emisor envía uno o varios flujos de control e inmediatamente pasa al estado destino indicado por dicha transición. En un instante posterior, otro proceso podrá ejecutarse, posiblemente ejecutando otra transición como consecuencia del flujo recibido. La relación de transición correspondiente tiene la forma:

$$\rho_\tau^A : (\pi = s) \wedge [c] \wedge (\pi' = s') \wedge env(a) \wedge rec(^\bullet P) \qquad (2)$$

donde:

$[c] = \bigwedge_{f_j \in C} (f_j = cond(f_j, \tau))$

$env(a) = \bigwedge_{f_j' \in A} (f_j' = acc(f_j, \tau))$

$rec(^\bullet P) = \bigwedge_{f_j' \in (C \cap F^a)} (f_j' = 0)$

que significa que cuando el proceso se halla en el estado local s y la transición se halla habilitada ($[c]$),en el siguiente instante de ejecución del sistema, éste llegará al estados', y enviará los flujos de control presentes en la acción ($env(a)$).

El término$rec(\bullet P)$, es denominado puesta a cero de los flujos asíncronos de entrada al proceso. Su significado se explica a continuación: Puesto que en el modelo matemático los eventos son representados como variables, es preciso que cuando un proceso envía un evento (modifica el valor de un flujo de evento dándole el valor uno), éste valor no permanezca permanentemente, sino que solamente sirva al proceso receptor para la notificación de dicho evento, tras lo cual vuelve a ser puesto a cero. Con ello se permite que un evento que llega a un proceso que se halla en un estado del cual no parte ninguna transición que contenga dicho evento en su condición, sea descartado.

Para cada estado localπ=s, es necesario añadir una transición adicional denominada transición nula $_N(s)$ que se habilita cuando ninguna de las transiciones que parten de s lo está, y cuya acción incluye solamente el término$rec(P)$:

$\rho_{\tau_N} : (\pi = s) \wedge \neg[c] \wedge (\pi' = s) \wedge rec(^\bullet P)$

2.2. Transiciones Síncronas

Una transición síncrona implica que el proceso emisor no completa la ejecución de la transición hasta que el receptor haya procesado el evento recibido. Una transición será síncrona cuando exista un flujo síncrono en la acción. La relación de transición correspondiente consiste en dos partes que se describen a continuación, utilizando la notación habitual para el proceso emisor, y la misma, pero con un subrayado (_) para el receptor.

En el primer paso, la relación de transición es similar a la asíncrona:

$\rho_\tau^E : (\pi = s) \wedge [c] \wedge (\pi' = \bar{s}') \wedge env(pre(a)) \wedge rec(^\bullet P)$

La diferencia es que el proceso emisor no es conducido al estado destinos' sino a un estado intermedio $\bar{s}$, denominado estado de sincronización, del cual no podrá salir hasta que el receptor procese el evento recibido.

El segundo paso representa la cooperación entre ambos procesos, y por tanto, implica la ejecución de una transición conjunta:

$$\rho_\tau^R : \begin{bmatrix} (\pi = \bar{s}) \wedge (\underline{\pi} = \underline{s}) \wedge [\underline{c}] \wedge \\ (\pi' = s') \wedge (\underline{\pi}' = \underline{s}') \wedge \\ env(post(a)) \wedge env(\underline{a}) \wedge rec(^\bullet P) \wedge rec(^\bullet \underline{P}) \end{bmatrix}$$

2.3. Modelo de Concurrencia

Las relaciones de transición anteriores definen la ejecución de una transición en un solo proceso (dos en el caso de comunicación síncrona). La semántica global del modelo se definirá componiendo las relaciones de transición correspondientes a todas las transiciones de todos los procesos, siendo preciso definir previamente el modelo de concurrencia.

La ejecución del modelo consiste en una serie de pasos en los cuales se ejecuta alguna transición. Según [14] se pueden distinguir dos tipos:

- ***Modelo simultáneo:*** la ejecución de todos los procesos es conjunta, es decir, más de una transición (más de un proceso) pueden ejecutarse simultáneamente.

- ***Modelo entrelazado:*** La ejecución de cada proceso es independiente, y solamente una transición (un proceso) se puede ejecutar en cada instante, teniendo en cuenta que cuando hay más de una transición habilitada en un proceso se selecciona una de ellas de forma no determinista.

El modelo entrelazado es más general para un sistema donde puede haber varios procesadores que se ejecutan a ritmos diferentes. Sin embargo, plantea problemas de pérdida de sincronización entre procesos ya que supone reducir la concurrencia al no determinismo. Para solucionarlo, se puede optar por dos alternativas:

- ***Utilizar reglas de ejecución*** como las descritas por Ward/Mellor [24] o en Statecharts [12], en las cuales un evento procedente del exterior (macro-paso) desencadena una cadena de interacciones internas (micro-pasos), de forma que no se procesa ningún evento externo hasta que haya cesado la interacción interna

- ***Utilizar comunicación síncrona***, como en los lenguajes derivados del CSP.

Esta última posibilidad es la que se ha utilizado en este trabajo, en combinación con la comunicación asíncrona, ya que, aunque difiere de las reglas de ejecución propuestas en [24], es más simple, permitiendo más fácilmente que el código generado a partir de la especificación reproduzca fielmente ésta.

3. Traducción del Modelo

Una vez definido formalmente el modelo, se realiza su traducción automática al lenguaje del comprobador de modelos SMV [14]. Para ilustrar la idea general se utilizará como ejemplo un proceso que en el estado localπ=s puede ejecutar una sola transición $\tau: s_1 \xrightarrow{v_{12}, c_{12}/a_{12}} s_2$, ($v_{12}$ es un flujo de variable). Se incluye además la transición nula:

$$\rho_\tau(s): (\pi = s_1) \wedge (v_{12} = 1) \wedge (c_{12} = 1) \wedge (\pi' = s_2) \wedge (a_{12}' = 1) \wedge (c_{12}' = 0)$$

$$\rho_N(s): (\pi = s_1) \wedge \neg((v_{12} = 1) \wedge (c_{12} = 1)) \wedge (\pi' = s_1) \wedge (c_{12}' = 0)$$

La traducción a SMV del conjunto de relaciones de transición de un proceso consiste en recorrer cada una de las variables de que consta la relación de transición en la parte correspondiente a $A_\tau(')$ y crear el código que permite asignar el valor correspondiente en función de $C_\tau()$. Para las relaciones de transición del ejemplo anterior, los fragmentos de código resultantes son:

1. Variable de estado local π':

```
next(pi) :=
  case
      (pi=s1) & v12 & c12 : s2;
      true : pi;
  esac;
```

que asigna el valor que tendrá la variable π (representada en el código por `pi`) en el siguiente instante de ejecución (`next`). Este valor será s_2 cuando el proceso se halla en el estado s_1 y además v_{12}=*true* y c_{12}=*true*. Si no se cumple lo anterior, la última línea de la sentencia `case` asignará a π el mismo valor que tenía (no hay cambio).

2. Flujos de la acción a_{12}' (de forma similar):

```
next(a12) :=
  case
   (pi=s1) & v12 & c12 : 1;
     true : a12;
  esac;
```

3. Puesta a cero de los flujos de entrada c_{12}'. No se realiza condicionalmente puesto que se ha de hacer siempre que se ejecute el proceso:

```
next(c12) := 0;
```

Todo lo anterior se realiza para cada proceso, encerrando el código correspondiente en un módulo de SMV. Los algoritmos que automatizan el proceso incluyen otros aspectos como los diferentes tipos de comunicación, transiciones con prioridades y selección no determinista [21].

4. Verificación

Las propiedades que se desean verificar se escriben en lógica temporal CTL [5]. En CTL se pueden escribir fórmulas que permiten expresar propiedades de seguridad y vivacidad. Así, si es una fórmula, también serán fórmulas las siguientes:

- **AG**(ϕ) que significa que ϕ será siempre cierta, para cualquier posible evolución del sistema (seguridad)
- **EF**(ϕ) que significa existe una posible evolución del sistema en el que ϕ será cierta en el futuro (posibilidad).
- **AF**(ϕ) que significa que para cualquier posible evolución del sistema, ϕ será cierta en el futuro (obligatoriedad).

SMV permite expresar el modelo tal y como se ha indicado en la sección anterior y añadirle una o varias fórmulas escritas en **CTL**. Tras la ejecución, indicará si las fórmulas son ciertas, o si alguna no lo es, proporcionará un contraejemplo indicando las causas, que será muy útil para depuración [20]. Sin embargo, el problema real es mucho más complicado puesto que cuando el sistema crece de tamaño, la solución del problema de verificación se convierte en imposible debido a la explosión de estados (el número de estados aumenta exponencialmente). Aunque los comprobadores simbólicos de modelos [4] intentan paliar el problema, no existe una solución universal. Es preciso, por tanto, utilizar determinados recursos para disminuir el número de estados del modelo que sirve de entrada al comprobador de modelos como la utilización de modelos simplificados y la verificación modular, los cuales se tratan en las secciones siguientes mediante un ejemplo.

4.1. Utilización de Modelos Simplificados

La simplificación del modelo es algo similar al proceso de prueba del software utilizando conductores y resguardos. Parte del modelo se sustituye por otro que sea más simple y además equivalente al anterior. Esta equivalencia ha de ser determinada en función del tipo de propiedades a verificar, puesto que el problema es radicalmente diferente si se trata de propiedades de vivacidad o de seguridad. Para estas últimas, el método que se propone está basado en lo siguiente.

Supóngase que existe un modelo M junto con su entorno E, y que se desea probar el cumplimiento de una propiedad de seguridad **AG**(ϕ) en la composición en paralelo de ambos [$M||E$]. Puesto que una propiedad de seguridad ha de ser cierta para cualquier posible camino (computación) del sistema, si se sustituye E por otro (E_s) de forma que el modelo [$M||E_s$] (denominado simplificado) generan un conjunto de computaciones que son superconjunto del modelo original, la propiedad que se demuestre como cierta en el modelo simplificado lo será en el modelo original. No así al contrario, puesto que puede ser falsa en el modelo simplificado a lo largo de una computación que no está presente en el modelo original. Todo ello se ilustra a continuación con un ejemplo real.

El ejemplo que se utilizará consiste en un sistema cuya misión es adquirir una muestra del carbón transportado por los camiones con destino a una central térmica [18], [19] *. El modelo esencial del sistema consta de 23 DFD's con 21 STD's y 12 tablas de decisión. En total, se estiman 10^{127} estados.

Primero se construye el grafo de procesos donde se representan en un diagrama plano todas las transformaciones de control del modelo (**Fig.3**). Las transformaciones de datos se discretizan para convertir los flujos de datos en flujos variable [21].

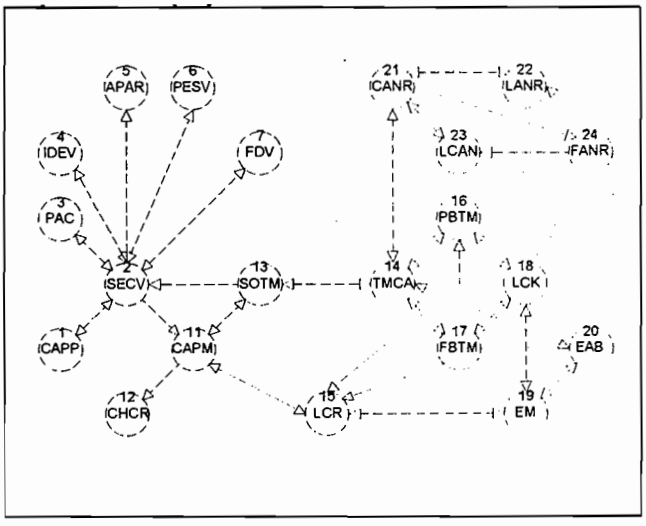


Figura 3: Grafo de Procesos

A continuación se escribe la propiedad a verificar. En este caso se desea probar una propiedad de seguridad muy importante que afirma que nunca se puede tomar muestra mientras el vehículo no está correctamente posicionado en la plataforma de toma de muestras. La importancia de dicha propiedad es debida a que el dispositivo que adquiere la muestra (denominado 'pincho') es un elemento muy pesado que puede provocar destrozos en la caja del camión o graves daños a una persona si ésta se halla en la plataforma. La fórmula correspondiente es:

$$\mathbf{AG}\neg(V.CicloTM \cap \neg V.VehiculoDentro) \quad (3)$$

Esta fórmula está expresada en función de intervalos, que son flujos de variable a través de los cuales se define el conjunto de estados en los que se puede encontrar el sistema entre la aparición de varios eventos. Estos intervalos, en función de los flujos de evento se representan como:

$$V.CicloTM = [DTMIniAUTO : DTMFinCiclo \vee DTMErrorCiclo] \quad (4)$$

$$V.VehiculoDentro = [PCPPosicionado : PCPDaSalida]$$

El siguiente paso consiste en marcar en el grafo de procesos los flujos de control que intervienen en la propiedad de seguridad y seleccionar un conjunto de procesos interconectados que incluyan éstos. El resultado es el tronco del grafo de procesos (**fig.4**), en el cual los procesos representados son *SECV* (SECuenciamiento de operaciones del Vehículo), *CAPM* (Control de Arranque y Parada de toma de Muestras), *SOTM* (Secuenciamiento de Operaciones de Toma de Muestras) y *TMCA* (Toma de Muestras en CArrusel). Supóngase que en un principio se eligen como pertenecientes al tronco los procesos representados en la **figura 4** excepto *CAPM*.

A continuación se suprimen todos los procesos que no pertenecen al tronco y se simplifican las variables que comunican procesos del tronco con procesos suprimidos. La simplificación de variables consiste en disponer éstas de tal forma que puedan tomar cualquier valor, con lo cual, el modelo resultante será menos determinista que el modelo completo, generando un conjunto de computaciones que son un superconjunto de éste. Seguidamente se pueden suprimir algunas transiciones que, tras la simplificación de variables, generan transiciones ociosas. Finalmente se realiza la traducción automática de los procesos pertenecientes al tronco, incluyendo la propiedad a verificar (3).

Tras ejecutar el comprobador de modelos se determina que propiedad es falsa. El contraejemplo muestra que *SOTM* envía *IniCicloTM* y por tanto comienza el ciclo de toma de muestra (*DTMIniAUTO*), violando la propiedad de seguridad.

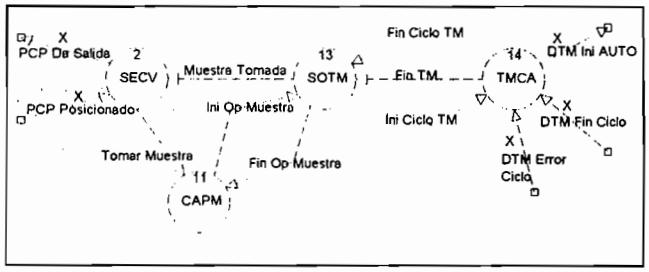


Figura 4: Tronco del Grafo de Procesos

La causa no es una incorrecta definición del modelo, sino la elección del tronco, ya que se ha eliminado el proceso *CAPM* que es quien ordena el comienzo del ciclo de toma de muestras cuando el vehículo se posiciona. Esto causa que *SOTM* pueda comenzar la toma de muestras en cualquier momento.

Es necesario, pues, reconsiderar la elección del tronco para incorporar más procesos y por tanto restringir la libertad de ejecución de éstos. Todo ello ha de ser dirigido por el conocimiento que tenga el ingeniero sobre el modelo, lo cual no es en principio un inconveniente, puesto que se supone que el proceso de verificación se realiza simultáneamente a la construcción del modelo. Se decide incluir *CAPM* y se repiten las operaciones anteriores. En este caso la fórmula también es falsa, pero el contraejemplo muestra ya las causas reales del fallo. Para su entendimiento se ha de tener en cuenta el STD del proceso *TMCA* que se representa en la **figura 5**, especialmente los estados correspondientes a la toma de muestras automática (parte izquierda del diagrama).

El contraejemplo muestra que al llegar un camión (se recibe el flujo *PCPPosicionado*), comienza un ciclo de toma de muestra mediante la ejecución en secuencia de *SECV*, *CAPM*, *SOTM* y *TMCA*. Cuando finaliza la toma de muestra automática se recibe *DTMFinCiclo* con lo cual *TMCA* envía *FinTM* y finalmente se produce *PCPDaSalida* que permite al camión salir de la plataforma. Pero mientras tanto *TMCA*, que todavía no ha finalizado su ciclo, se encuentra en el estado *ComprobandoBoteTM*, en el cual está esperando a que la muestra obtenida se deposite correctamente en el recipiente correspondiente. Puesto que esta operación puede fallar (por ejemplo, debido a que la cantidad de carbón adquirida es insuficiente), es posible que finalice con error y que se repita un nuevo ciclo de toma de muestra, poniendo el pincho a trabajar mientras que el vehículo está fuera. Para resolverlo habrá que transformar parte del comportamiento de *TMCA* impidiendo dicha situación, repitiendo el proceso hasta que la fórmula sea cierta.

4.2. Verificación Modular

El método anterior de verificación tiene un problema debido a que es posible que el tronco del grafo de procesos sea suficientemente grande como para que la explosión de estados haga la verificación imposible. Ello hace necesario plantear la posibilidad de verificar propiedades parciales en partes del modelo y concluir una propiedad general, lo que se denomina verificación modular. Para ello se utiliza el teorema de composición de Abadi/Lamport [1] que se representa por la siguiente regla de inducción:

$$(5)\quad \frac{M_1 \text{Æ} \text{Í}_{\phi_{E1}} \phi_{M_1}\ ,\quad M_2 \text{Æ} \text{Í}_{\phi_{E2}} \phi_{M_2} \qquad \phi_{M_1} \rightarrow \phi_{E_2}\ ,\quad \phi_{M_2} \rightarrow \phi_{E_1}}{[M_1 \| M_2] \text{Æ} \text{Í} (\phi_1 \wedge \phi_2)}$$

La regla afirma que para probar en un modelo $M::[M_1 \| M_2]$ una propiedad expresada por $\phi = \phi_{M1} \phi_{M2}$ basta verificar que cada ϕ_{Mi} es cierta respectivamente en los componentes M_i cuando se cumple la suposición ϕ_{Ei} del entorno de cada M_i. Además se ha de cumplir que cada componente M_i garantiza la suposición ϕ_{Ej}, ij que efectúa el otro componente sobre su entorno.

Si no se hace ninguna suposición sobre el entorno ($\phi_{Ei}=true$), la expresión puede quedar reducida a:

$$\frac{M_1 \text{ sat } \phi_1, \quad M_2 \text{ sat } \phi_2}{[M_1 || M_2] \text{ sat } (\phi_1 \wedge \phi_2)} \tag{6}$$

En el ejemplo anterior se ha comprobado la propiedad (3) para el modelo esencial, pero esto no es suficiente, puesto que los flujos en base a los cuales está expresada no son flujos reales, sino abstracciones del comportamiento de un dispositivo que en el modelo esencial se representa como un terminador (*DTM*). Si se desea realizar el modelo de implementación sin distorsión del modelo esencial, se requiere construir un nuevo modelo para el terminador que constituya el enlace entre los flujos de control 'esenciales' y las señales 'reales' que maneja el dispositivo. Se pueden establecer entonces una serie de modelos que se comunican entre sí como se representa en la **figura 6**.

El modelo esencial (*M.E.*) ya ha sido desarrollado y verificado por separado. El modelo físico del dispositivo (terminador *Pincho* en la figura 6) se ha de construir de acuerdo con el comportamiento de éste (se trata de un PLC cuya programación no es modificable). Resta por desarrollar el modelo para el 'terminador' (*DTM*). Una vez realizado éste, se puede plantear la propiedad deseada para el conjunto global, es decir para todo el sistema:

$$\mathbf{AG} \neg(V.PinchoOn \wedge \neg V.VehiculoDentro) \tag{7}$$

donde *V.PinchoON* es una variable digital (física).

Para el modelo del terminador (proceso*DTM* en la Figura 6) y el dispositivo físico se puede probar

$$\mathbf{AG} \neg(V.PinchoOn \wedge \neg V.CicloTM) \tag{8}$$

Por el teorema de Lamport simplificado (6) se concluye que se cumple (3)(8). Puesto que se puede comprobar que (3)(8)(7) es cierto (basta con realizar una simple tabla de verdad), se puede afirmar entonces que la propiedad deseada (7) se cumple para el modelo global.

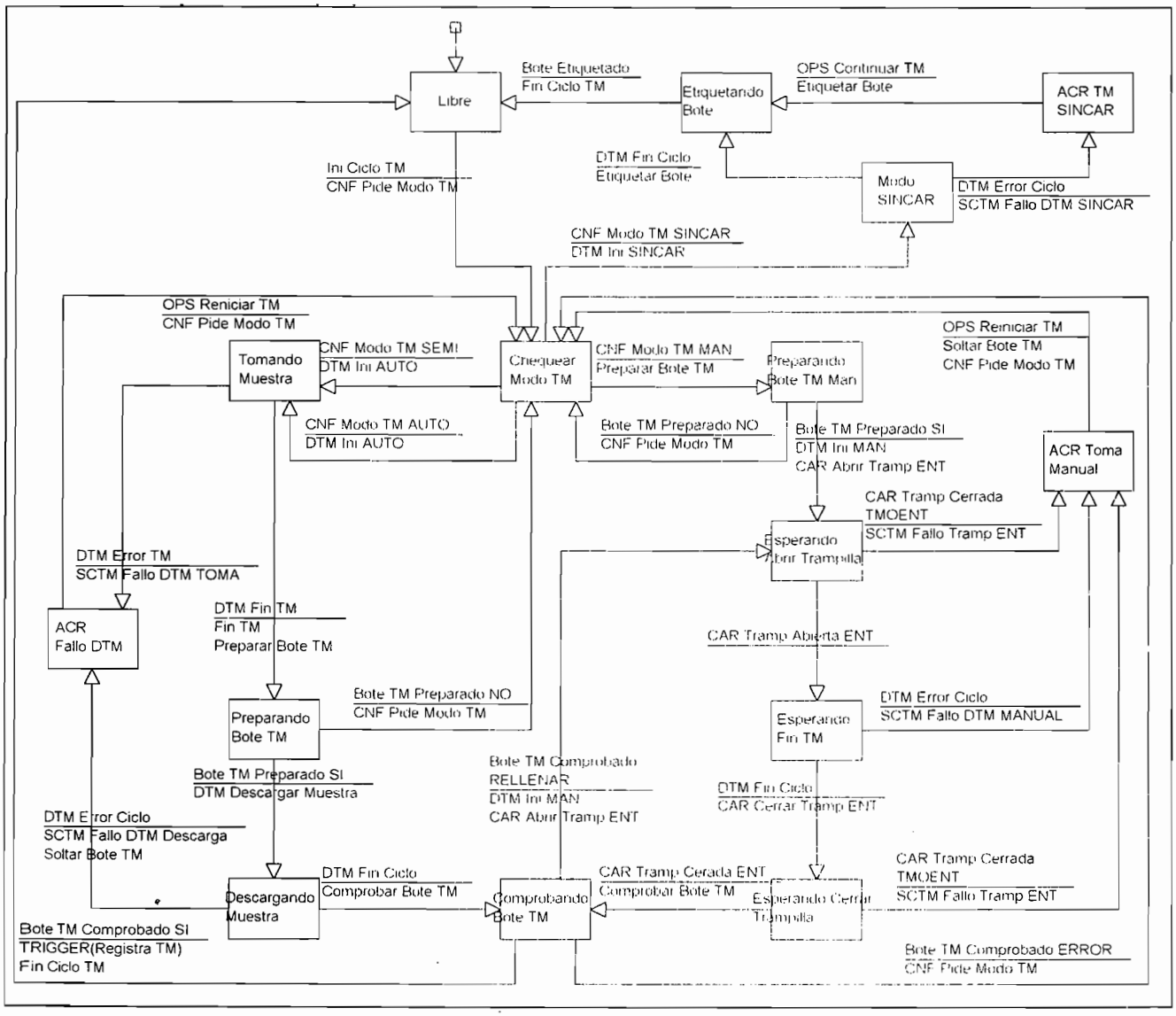


Figura 5: Toma de Muestras

4.3. Propiedades de Vivacidad

Las propiedades de vivacidad son mucho más complejas de probar que las de seguridad puesto que implican que 'algo sucederá', y por tanto, no son posibles las mismas simplificaciones que se han utilizado anteriormente. Por ejemplo, una propiedad interesante será que siempre que se posiciona un vehículo, se le podrá dar salida, es decir, que el sistema no quedará bloqueado en algún estado desde el cual es imposible dar salida:

$$\mathbf{AG}(PCPPosicionado \rightarrow \mathbf{EF}(PCPDaSalida))$$

Para realizar la verificación no se pueden suprimir procesos puesto que estos pueden influir en el bloqueo del sistema. Basta con que uno se bloquee para que se bloquee todo el sistema. No obstante, la prueba de propiedades de vivacidad puede realizarse de forma parcial, permitiendo detectar problemas que no son tan evidentes tras una simple inspección de la especificación.

Por ejemplo, para el conjunto de modelos correspondientes al terminador y al modelo físico del dispositivo de toma de muestras, se pueden afirmar, entre otras, propiedades como las siguientes:

$$\begin{aligned} &\mathbf{AG}\,(DTMIniAuto \rightarrow \mathbf{AF}\,(DTMFinTM \\ &\qquad\qquad \vee DTMErrorTM\ )) \\ &\mathbf{AG}\,(DTMIniAuto \rightarrow \mathbf{EF}\,(DTMFinTM\ )) \\ &\mathbf{AG}\,(DTMIniAuto \rightarrow \mathbf{EF}\,(DTMErrorTM\ )) \end{aligned} \tag{9}$$

Ejecutando el comprobador de modelos (en este caso sin simplificaciones) se determina cuáles son ciertas y cuáles son falsas, pudiendose depurar el modelo examinando el contraejemplo.

Pero la prueba de las propiedades anteriores no garantiza que cada una de las peticiones realizadas con *DTMIni* sea atendida. Ello es debido a que los flujos de control son transformados en variables, por lo que es posible que el flujo que indica el inicio del ciclo sea enviado más de una vez antes de haber recibido el que indica finalización (*DTMFinTM* o *DTMErrorTM*), ignorando la segunda petición.

Para garantizar que el modelo atenderá a cada una de las peticiones, habrá que garantizar que no se envía una petición hasta que no se haya satisfecha la anterior, lo cual se puede expresar como una propiedad de seguridad de la siguiente forma:

$$\mathbf{AG}\ \neg(\tau_a(DTMIniAuto\ ) \wedge V.Pincho\ )$$

$$V.Pincho\ = [DTMIniAuto\ : DTMFinCiclo \vee DTMErrorCiclo\ \vee DTMErrorTM\ ]$$

donde se utiliza la notación $_a(f)$ para referirse al instante en que aparece el flujo *f*. Su traducción a SMV se realiza utilizando una variable que cambia de valor solamente durante un paso de ejecución del sistema.

5. Conclusiones

La utilización de los métodos estructurados SA/RT permite un desarrollo más disciplinado de este tipo de sistemas, y facilita su mantenimiento. Esto, combinado con la utilización de comprobadores de modelos para verificar determinadas propiedades permite aumentar el grado de corrección de la especificación con facilidad.

En este artículo se han mostrado las posibilidades de este enfoque. Las descripciones detalladas de la sintaxis y semántica, algoritmos de traducción, método de simplificación de procesos y variables, así como otros ejemplos se pueden encontrar en [21].

El método desarrollado para la simplificación del modelo puede plantear el problema de no encontrar una solución al mostrar una propiedad como falsa sin causa aparente, siendo necesario añadir más procesos al tronco, y causando que el modelo sea suficientemente grande como para que no pueda ser verificado con los recursos computacionales existentes. Para solucionar el problema se puede usar la verificación modular. No obstante, esto puede ser síntoma de un exceso de interelaciones (acoplamiento) entre procesos, lo cual obligará a reestructurar la especificación haciéndola más simple.

Entre los aspectos a investigar a corto plazo se encuentra el acercamiento de la semántica elegida a las reglas de ejecución propuestas por Ward/Mellor o alguna de las diferentes semánticas de Statecharts, lo cual reducirá los problemas causados en ocasiones por el no determinismo de la especificación.

La adopción de una serie de reglas de ejecución para incorporar a la semántica y la consiguiente traducción, así como la definición de prioridades en los procesos es el objetivo inmediato del trabajo actual. Asimismo, es importante el desarrollo de un soporte automatizado para el método de simplificación, ya que hasta ahora se realiza la traducción automática, pero sería deseable poder interactuar de forma gráfica para seleccionar el grafo de procesos y animar los contraejemplos, así como poder realizar automáticamente verificaciones de regresión cuando algún componente del modelo cambia.

En otra linea de trabajo, se encuentra la lucha contra la explosión de estados, por una parte la optimización de los comprobadores de modelos en aspectos críticos como el ordenamiento de las variables [2], los métodos de reducción composicional [17] y la utilización de los métodos de verificación modular [1] para propiedades de vivacidad, ya que son válidos siempre que las suposiciones que realiza un proceso sobre su entorno sean expresables como propiedades de seguridad.

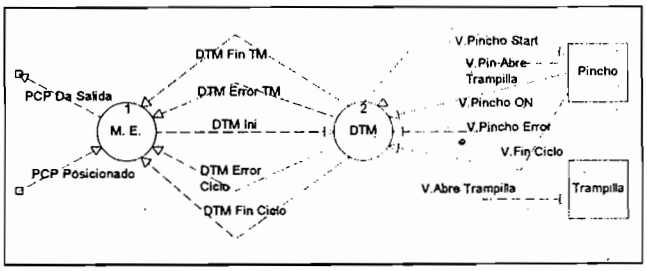


Figura 6: Composición Modelo Esencial y su implementación

6. Referencias

[1] **Abadi, M., Lamport, L.:** *Conjoining Specifications.* DEC Software Research Centre. Research Report no. 118 , 1993.

[2] **Aziz, A., Tasiran, S., Brayton, K.:** *BDD Variable Ordering for Interacting Finite State Machines.* University of California, Berkeley. Tech. Report UCB/ERL M93/71, 1993.

[3] **Bryant, R. E.:** *Graph-Based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers, Vol. 35, no. 8, August, pp. 677-691, 1986.

[4] **Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., Hwang, L. J.:** *Symbolic Model Checking: 12^{20} States and Beyond.* Information 1020 and Computation, Vol. 98, pp. 142-170, 1992.

[5] **Clarke, E. M., Emerson, E. A., Sistla, A. P.:** *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications.* ACM Transactions on Programming Languages and Systems, Vol. 8, no. 2, April, pp. 244-263, 1986.

[6] **Craigen, D., Gerhart, S., Talston, T.:** *Experience with Formal Methods in Critical Systems: Regulatory Case Studies.* IEEE Software, Vol. 11, no. 1, January, pp. 31-40, 1994.

[7] **Elmstrom, R., Lintulampi, R., Pezzé, M.:** *Giving Semantics to SA/RT by Means of High-Level Timed Petri Nets.* Real-Time Systems Journal, Vol. 5, no. 2/3, pp. 249-271, 1993.

[8] **Felder, M., Mandrioli, D., Morzenti, A.:** *Proving Properties of Real-Time Systems Through Logical Specifications and Petri Net Models.* IEEE Transactions on Software Enginnering, Vol. 20, no. 2, pp. 127-141, 1994.

[9] **Harel, D.:** *STATECHARTS: A Visual Formalism for Complex Systems.* Science of Computer Programming, North Holland, Vol. 8, pp. 231-274, 1987.

[10] **Hatley, D. J., Pirbhai, I.:** *Strategies for Real Time System Specification.* Dover Press, New York, 1987.

[11] **Leveson, N. G., Turner, C. S.:** *An Investigation of the Therac-25 Accidents.* IEEE Computer, vol. 26, no. 7, pp. 18-41, 1993.

[12] **Leveson, N. G., Heimdahl, M. P. E., Hildreth, H., Reese, J. D.:** *Requirements Specification for Process Control Systems.* IEEE Transactions on Software Engineering, Vol. 20, no. 9, pp. 684-707, 1994.

[13] **Manna, Z., Pnueli, A.:** *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1991.

[14] **McMillan, K. L.:** *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD. Thesis, Carnegie Mellon University (Tech. Report CMU-CS-92-131), 1992.

[15] **Ostroff, J. S.:** *Formal Methods for the Specification and Design of Real-Time Safety Critical Systems.* Journal of Systems and Software, April, pp. 33-60, 1992.

[16] **Richter, G., Maffeo, B.:** *Towards a Rigorous Interpretation of ESML - Extended Systems Modelling Language.* IEEE Transactions on Software Engineering, Vol. 19, no. 2, Februray, pp. 165-180, 1993.

[17] **Simone, R., Ressouche, A.:** *Compositional semantics of Esterel and verification by compositional reductions.* Proceedings of the Computer Aided Verification Conference, Springer-Verlag LNCS 818, 1994.

[18] **Tuya, J., Sánchez, L., Zurita, R., Corrales, J. A.:** *A Pragmatic Task Design Approach Based on a Ward/Mellor Real-Time Structured Specification.* Proceedings of the 4th European Software Engineering Conference. pp. 301-312, 1993.

[19] **Tuya, J., Sevilla, I., Sánchez, L., Corrales, J. A.:** *Using Structured Methods in the Development of the User Interface Subsystem for a Reactive System.* 6th International Conference of Advanced Information Systems. In Poster Outlines, pp. 14-16, 1994.

[20] **Tuya, J., Sánchez, L., Sevilla, I., Corrales, J. A.:** *Verificación de Sistemas Reactivos Utilizando Lógica Temporal: Un Caso Práctico.* Actas de la XVI Escuela de Verano de Informatica, Asociacion Española de Informatica y Automatica, pp. 55-68, 1994.

[21] **Tuya, J.:** *Especificación y Verificación de Sistemas Reactivos Utilizando Métodos Estructurados y Lógica Temporal.* Tesis Doctoral, Universidad de Oviedo, 1994.

[22] **Tuya, J., Sánchez, L., Corrales, J. A.:** *Using a Symbolic Model Checker for Verify Safety Properties in SA/RT Models.* Proceedings of the 5th European Software Engineering Conference. pp. 59-75, 1995.

[23] **Ward, P., Mellor, S.:** *Structured Development for Real-Time Systems.* Prentice-Hall, 1985.

[24] **Ward, P.:** *The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing.* IEEE Transactions on Software Engineering, Vol. 12, no. 2, pp. 198-210, 1986.

* Proyecto financiado por el Patronato de Industrias Eléctricas (PIE 031/017), Compañía Eléctrica de Langreo (CEL 053) y Universidad de Oviedo (D3-029-90).

Ingeniería del Software

Impacto en la calidad de la fase de pruebas con el uso de técnicas formales

Gabriel Huecas (1), José A. Mañas (2), Tomás Robles (2)
(1) Dpt. Informática. Univ. Alfonso X el Sabio, ***gabriel@uax.es***
(2) Dpt. Ing. Telemática, E.T.S.I. Telecomunicación, ***jmanas@dit.upm.es***

Este trabajo ha sido parcialmente subvencionado por el proyecto E-MEDAS.

Resumen: *El surgimiento de normas por parte de organismos normalizadores de Servicios y Sistemas de Comunicaciones, como ISO e ITU supone un cambio en la mentalidad tanto en el desarrollo como en la producción para los fabricantes de sistemas de comunicaciones; normas que los fabricantes deben cumplir y organismos independientes deben certificar u homologar. En este entorno, las Técnicas de Descripción Formal son un mecanismo clave para el diseño y especificación de dichos protocolos. En este artículo se presentan las áreas de impacto donde las FDTs suponen una mejora en la calidad y productividad. Para ello, se conceptualiza y formaliza el proceso de ejecución de Pruebas de Conformidad y elementos integrantes en las arquitecturas de pruebas. Además, se definen y formalizan los componentes de las pruebas. Todo ello se contextualiza en el entorno definido por las normas 9000 de ISO, donde se definen los parámetros que definen la calidad de productos ISO. Para ello, se describe el impacto de esta formalización y el uso de las FDTs en los parámetros que definen la calidad de productos ISO.*

1. Introducción

En la actualidad, las Técnicas Formales de Descripción (FDTs) han tomado un papel fundamental en la descripción de sistemas de forma precisa, concisa y sin ambigüedades. Como resultado de ello se disponen de Especificaciones Formales de sistemas de telecomunicación que pueden ser tratadas automáticamente por herramientas software. En principio, estas especificaciones tienen como objeto mejorar la fase de diseño y facilitar el entendimiento entre el cliente y el diseñador. Sin embargo, el hecho de disponer de tales especificaciones ha generado multitud de teorías, metodologías y algoritmos que intentan explotarlas en los campos de Generación y Ejecución de Pruebas, Entornos Transformacionales, Simuladores, Generadores de Prototipos. Esto ha provocado un gran impacto en el mercado de servicios y sistemas, que evoluciona hacia un soporte automatizado en todas las fases del ciclo de vida. Particular interés tiene la fase de ejecución de pruebas: al terminar el producto, éste se ve sometido a una serie de pruebas con diferentes objetivos: por un lado, se desea que el producto cumpla unos mínimos de calidad. Por otro lado, el cliente está interesado en comprobar que dicho producto cumple cierto requisitos, normalmente, que se atienen a una cierta norma. Asimismo, se ejecutan pruebas no funcionales, como pruebas de prestaciones, robustez, etc. En este artículo nos vamos a centrar en las denominadas *pruebas de conformidad,* cuyo objetivo es garantizar que el producto se atiene a una norma. Para lograr este objetivo, la implementación es probada respecto de su especificación de referencia. Está demostrado [Dil91] que los productos homologados tienen una alta probabilidad de *interoperar* con otros, y que esta probabilidad es menor para productos no conformes a la misma norma (Evidentemente, ni siquiera se intenta que interoperen productos que no han sido fabricados para ello).

ISO ha normalizado una metodología y un entorno específico de ejecución de pruebas de conformidad [ISO91] . La idea básica es disponer de baterías de pruebas para cada norma, disponible públicamente para fabricantes y usuarios. Estas pruebas podrán ser desarrolladas manualmente o bien derivadas de forma automática o semi-automática a partir de la especificación formal de la norma. Puntos claves de esta norma son la *reusabilidad* y *flexibilidad* de las Baterías de Pruebas.

Por otro lado, ISO define la serie de normas 9000, con el objeto de identificar los parámetros que definen la calidad de productos e identificar que aspectos influyen en dichos parámetros.

2. Objetivos

El objetivo de este artículo es presentar un modelo de ejecución de pruebas genérico, en las que pruebas descritas mediante FDTs son ejecutadas contra productos reales. Dado que se tratan de Pruebas de Conformidad, la Implementación Bajo Pruebas (IUT) será tratada como una caja negra que se somete a experimentación. A cada ejecución de una prueba se le asigna un veredicto que etermina si se cumplió el propósito para el cual fue diseñada.

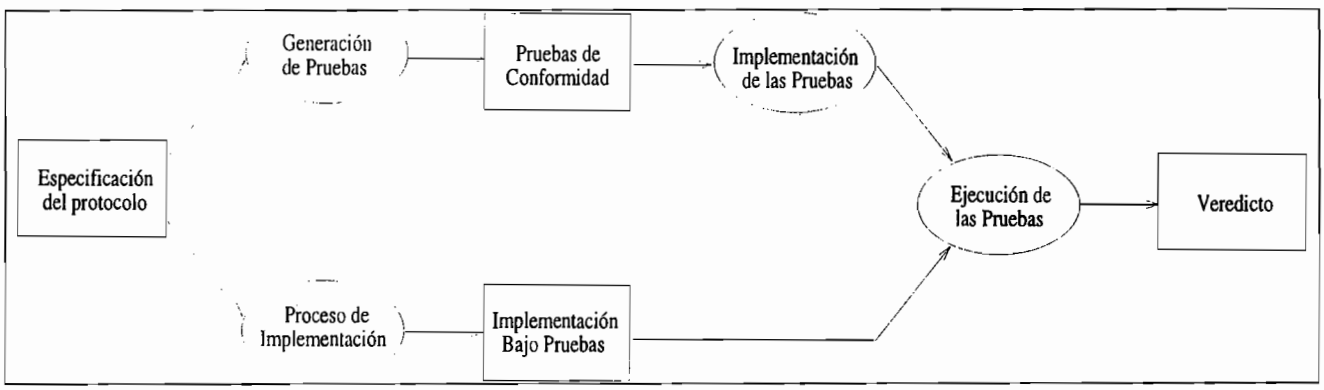


Figura 1. Proceso de Certificación de Conformidad

El sistema utilizará la especificación de referencia con el propósito de poder informar al fabricante de los errores hallados, expresados en términos de dicha especificación. Ahora bien, el fabricante es libre de elegir la forma en que se accede a su producto. Por lo tanto, como paso previo, se debe conseguir un modelo operativo de cada ejecución, que abstraiga los elementos comunes de la ejecución de pruebas y que minimice la adaptación del sistema de ejecución a cada nueva IUT. De esta forma, se mejora la *reusabilidad* y la *fiabilidad* de la fase de ejecución de pruebas.

3. Entorno y Metodología OSI

El proceso de garantizar la conformidad de una implementación respecto de su especificación propuesto por ISO se muestra en la **figura 1.**

La primera fase es generar una batería de pruebas (*Test Suite*) para un protocolo (OSI) en particular. Esta fase se la conoce como Generación o Derivación de Pruebas. Existen serios intentos de automatizar esta fase, como por ejemplo [Rob91]. La Batería de Pruebas generada es abstracta, en el sentido de que es completamente independiente de la implementación. En ISO, típicamente esta batería estará expresada en el lenguaje TTCN [ISO91].

Como esta batería es genérica, habrá una subfase de *Selección de Pruebas,* puesto que el fabricante habrá optado por una serie de opciones de la norma, dejando de lado otras. Esta información la proporciona el fabricante en un documento conocido como PICS (Protocol Implementation Conformance Statement [ISO91]).

La segunda fase consiste en la realización de los medios de ejecución de pruebas, es decir, *Implementar las Pruebas.* Los *Casos de Prueba* son transformados en *Pruebas Ejecutables*, que serán particulares para cada sistema de pruebas. Esta particularización dependerá de los detalles de implementación de los eventos representados de forma abstracta en la especificación de referencia. Existe un documento, llamado PIXIT (Protocol Implementation eXtra Information for Testing [ISO91]), donde el fabricante incluye toda la información necesaria para poder ejecutar estas pruebas. En definitiva, este documento describe las interfaces físicas del producto, así como detalles de codificación de datos, etc.

La última fase es la *Ejecución de Pruebas*. El producto es ejercitado de acuerdo a los *Casos de Prueba Ejecutables*, con el objetivo de emitir un *veredicto*. Este veredicto determina la consecución del objetivo para el cual se desarrolló la prueba. El veredicto puede ser de PASA si el objetivo se ha cumplido, FALLA si no se cumple e INCONCLUSO si no es ninguno de los anteriores (este caso puede darse cuando en un sistema se ejecuta una secuencia *legal* de acciones pero que aparta al sistema de *verificar* la consecución del objetivo de la prueba.. Este resultado es documentado en un *Informe de Pruebas de Conformidad del Protocolo.*

4. Casos de Prueba

La obtención de la batería de pruebas es una fase previa orientada a organizar técnica y administrativamente la realización de las pruebas, con el objeto de facilitar el *mantenimiento* de las mismas.

Los **casos de prueba** constan de un ***preámbulo***, un ***cuerpo*** y un ***postámbulo*** [ISO91]. El preámbulo es una fase preliminar en la que la implementación se lleva al estado en el que se quiere realizar la prueba. En pruebas muy sencillas, este preámbulo es nulo. El cuerpo es el objeto en sí de la prueba. Los preámbulos de pruebas complejas son el cuerpo de pruebas preliminares. Por último, el postámbulo es el conjunto de acciones que nos permiten llevar la implementación a un estado reconocible tras una prueba. A veces es tan simple como un botón de *reset*. De acuerdo con la definición de conformidad, solamente trazas contempladas en la especificación serán probadas sobre el producto. La selección de casos de prueba se reduce a extraer trazas posibles de la especificación de referencia.

Propósitos de Prueba. Las pruebas se organizan por propósitos. Estos consisten en identificar un estado inicial, al que llega gracias al preámbulo, y un objetivo que es el cuerpo de la prueba. Un propósito de prueba es *"una descripción en prosa de un objetivo de prueba definido con precisión, centrado en un único requisito de conformidad, de acuerdo con la especificación de la norma correspondiente"* [ISO91].

5. Ejecución de pruebas

La ejecución de una prueba consiste en evolucionar en paralelo la IUT y la prueba. Hay que tener en cuenta, además de los aspectos operacionales, que mientras que la IUT sólo ejecuta una posible secuencia de acciones, sobre la prueba se pueden seguir diferentes trazas en un momento dado. Evidentemente, si la prueba está bien diseñada, será posible decidir, tarde o temprano, cuál de las trazas está siendo realmente observada.

Por tanto, el Ejecutor de Pruebas debe encargarse de decidir si estimula o espera la reacción de la IUT. Si la prueba ha terminado, emitirá el veredicto adecuado. En caso contrario, deberá determinar cual es el nuevo evento a ofertar/observar. Por ello tendrá que seguir los diferentes estados de la prueba. Además, tendrá en cuenta aspectos organizativos como la división entre preámbulo y cuerpo de la prueba, y el posible bloqueo de la IUT.

5.1. Temporizadores

Hay que prever que la implementación falle y que durante las pruebas algún evento previsto en las trazas no se lleve a cabo. La decisión de que un evento no ocurre debe tomarse en base al vencimiento de un temporizador: 'si no ocurre en T segundos, considérese que no va a ocurrir nunca'. Esta información sobre tiempos de espera va estrechamente asociada a los propósitos de prueba, pudiéndose expresar bien en términos globales (idéntico criterio para todos los eventos) o particulares (usando temporizadores específicos para cada caso de prueba o, incluso, para algunos eventos).

5.2. Clases de Pruebas

En la fase de generación de la batería de pruebas se hace un análisis exhaustivo de la especificación buscando trazas que recorran los objetivos de pruebas. De este análisis resultan los casos de prueba clasificados en tres clases:

Obligatorios. Una cierta traza de prueba es de obligado cumplimiento. Es el grupo ideal y más simple de tratar. U ocurre, o se rechaza el producto.

Opcionales. Una cierta traza de prueba es opcional, siendo decisión del fabricante implementarla o no. Resultan casos

de prueba parametrizados por valores que sólo se conocerán al ir a pasar las pruebas. Estos valores los proporcionará el fabricante en un documento conocido como PICS (Protocol Implementation Conformance Statement [ISO91]). El operador introducirá los valores concretos del PICS para un cierto producto. Un cierto número de pruebas puede quedar anulado en base a estos valores. Típicamente, las pruebas asociadas a cierto parámetro del PICS suelen estar agrupadas.

Indeterministas. Es muy frecuente en protocolos de comunicaciones que un cierto objetivo no se pueda alcanzar, sin que ello implique un error en la implementación. Por ejemplo, una desconexión en un protocolo orientado a conexión, puede impedir que se observe la entrega de un dato; pero la desconexión iniciada por el medio es un comportamiento correcto contemplado en la norma (otro tipo de pruebas, por ejemplo las de prestaciones, pueden fallar si, por ejemplo, el número de desconexiones iniciadas por el medio es escandalosamente elevado, impidiendo la transferencia efectiva de información; pero aquí nos estamos centrando en pruebas de conformidad, que no entran en aspectos de prestaciones).

Debido a comportamientos no deterministas de la especificación, puede ocurrir que, al intentar ejecutar la prueba, el producto actúe de forma legal (según lo especificado) pero apartándose del propósito de la prueba. La prueba contempla estos indeterminismos dirigiendo la implementación a un estado estable pero, evidentemente, no se puede asegurar que el propósito ha sido cumplido. Por ello, se clasifican las pruebas dependiendo de si se puede asegurar el cumplimiento de su objetivo. Esta clasificación, ampliada de [dNH84] es:

- MUST ACCEPT La especificación es determinista en el comportamiento referente al objetivo de la prueba, por lo que la IUT debe aceptar, en todo momento, los eventos de la misma.

- MAY ACCEPT La especificación admite comportamientos que se apartan del objetivo de la prueba, por lo que su terminación puede no asegurar el cumplimiento de su propósito.

- MUST REJECT Igual que MUST, pero la intención es que se rechace.

- MAY REJECT Igual que MAY, pero con la intención de que se rechace.

5.3. Relación entre la formalización y su concreción

Todo el mecanismo de descripción formal se basa en abstraer detalles concretos de la realidad, darles un nombre simbólico, y manipular este mundo de símbolos. Todo esto es muy cómodo para realizar manipulaciones simbólicas sobre la especificación; pero tiene un límite que se alcanza cuando hay que interactuar con un producto.

La correlación entre el mundo abstracto y el concreto se realiza, en nuestro sistema, en base a un mecanismo de anotaciones (básicamente, una anotación es un comentario especial con significado para ciertas herramientas [MdM89]) que permite asociar aspectos externos a eventos simbólicos. Hay básicamente dos casos a contemplar: eventos que ocurren en el mundo porque la especificación así lo desea (salientes), y elementos que ocurren en el mundo y la especificación constata que han acaecido (entrantes). ISO ha propuesto un lenguaje de especificación de pruebas (TTCN [ISO91]) que contempla esta clasificación explícitamente. Los eventos salientes se denotan por **!**, y los entrantes por **?**. Asimismo, SDL y ESTELLE definen los eventos que son salientes y entrantes. Desgraciadamente, LOTOS abstrae este 'detalle', lo que requiere que sea introducido posteriormente.

5.4. Imposición versus Observación

¿Quién decide qué evento ocurre, el probador o el producto sujeto a pruebas? La respuesta depende de la estructura de la especificación. Esta ya fue analizada durante la fase de generación de pruebas, y la información pertinente consta en el caso de prueba. Si en un estado dado el caso sólo prevé un posible evento, la respuesta es simple: no hay nada que elegir, u ocurre ese evento o algo ha fallado (se suele decir que hay bloqueo, aspecto que trataremos más adelante). Si el caso de prueba permite una decisión indeterminista, el probador debe estar preparado para observar cualquiera de las posibilidades y emitir el veredicto correspondiente.

5.5. Bloqueo

Los casos de prueba sólo contemplan terminaciones exitosas o inconclusas. Pero en cualquier momento puede ocurrir cualquier evento no previsto o, más precisamente, que ninguno de los eventos previstos ocurran dentro del plazo marcado en el propósito de prueba. Cuando un temporizador salta decimos que se ha producido un **bloqueo**.

Un bloqueo puede ocurrir durante la ejecución del preámbulo o durante la ejecución del cuerpo de la prueba. En el primer caso se asigna un veredicto de INCONCLUSO, siguiendo la costumbre de los centros de homologación y las recomendaciones de la ISO [ISO91]. Esta asignación es arbitraria y, usualmente, conlleva la repetición de la prueba un cierto número de veces. En cuanto pasa una vez, se considera pasada y no se repite más; pero si al cabo de un número predeterminado de intentos sigue fallando el preámbulo, se le asigna un veredicto final de INCONCLUSO. Esto también es arbitrario; pero sigue siendo práctica habitual.

Muy diferente resulta la situación en la que el probador se bloquea durante la ejecución del cuerpo de la prueba. En estos casos, el veredicto es inequívocamente FALLO. Se emite un informe del estado de la especificación de referencia en el momento del bloqueo, y se rechaza el producto (a efectos prácticos, puede que se intenten algunas pruebas más para aprovechar la sesión de homologación e intentar descubrir más errores antes de devolver el producto al fabricante; pero pase lo que pase con cualquier otra prueba, el producto será rechazado).

5.6. Prueba Correcta

Definimos como *prueba correcta* aquella que cumple:
- Cada caso de prueba debe incluir información acerca de la consecución de su propósito. éste debe ser único.
- La terminación de una prueba irá marcada según el comportamiento esperado: PASA o FALLA. Estos casos corresponden a pruebas de aceptación o de rechazo. Se incluye la terminación de INCONCLUSO para tratar el posible no determinismo de la especificación.
- En cada estado de la prueba o bien existe un único evento *imponible* o uno o varios *observables*.

- ***imponibles:*** un evento imponible es aquel que la prueba trata de forzar su ejecución por la implementación. Si ésta lo rechaza se considera que le ejecución ha sido bloqueada.
- ***observables:*** un evento observables es aquel que la prueba se limita a observar en su ejecución por parte de la IUT. Cuando ésta ejecuta uno de tales eventos la prueba se limita a evolucionar acordemente, sincronizando adecuadamente e intercambiando datos si fuera menester.
- Cada traza de la prueba que termine deberá identificar un único estado sobre la especificación, con el propósito de asegurar que la asignación de cobertura se hace sobre partes ejercitadas de la especificación.

5.7. Aspectos Operacionales y FDTs

En una prueba se ha de marcar el punto donde termina el preámbulo y comienza el cuerpo. Además, hay que tener en cuenta que la prueba puede no tener preámbulo. Por ello la anotación END_OF_PREAMBLE indicará que el preámbulo ha terminado.

El último evento de la prueba debe adjuntar la etiqueta de terminación. Se proponen las anotaciones PASA, FALLA e INCONCLUSO para los tres veredictos asignables, con las restricciones detalladas en la sección 5.6.

Hemos visto que los eventos serán imponibles u observables. Los lenguajes TTCN, SDL y ESTELLE indican la direccionalidad de los eventos externos. No así LOTOS, que requiere haber caracterizado el interfaz con anotaciones, mediante las anotaciones sobre la interfaz IN (imponible) y OUT (observable).

Por último, se consideran los bloqueos. Evidentemente, los debidos a reacciones erróneas de la IUT se detectan por la propia semántica de pruebas: al no ser una acción incluida en la prueba se determinará que es un bloqueo. Los debidos a falta de reacción por la IUT se resuelven con un temporizador. En SDL, ESTELLE y TTCN se dispone de primitivas de temporización. En LOTOS será introducido en el interfaz mediante la anotación **timeout #.** Tras # unidades de tiempo desde la última acción sin actividad, se considerará que se ha producido un bloqueo.

5.8. Asignación de veredictos

La asignación de veredictos [Rob91] descrita en la norma IS-9646 se resume en la **tabla 1**, relacionando la terminación de la ejecución de la prueba (filas) con la clasificación de la misma (columnas). En la primera columna se presentan las terminaciones de la ejecución de una prueba. Si existe un bloqueo en el preámbulo, el veredicto es INCONCLUSO. Si ocurre ya en el cuerpo principal de la prueba obtenemos normalmente un FALLO para pruebas de aceptación, excepto en el caso de prueba MUST REJECT, que es precisamente la intención de la prueba. Las tres últimas filas se refieren a la terminación de la prueba. Si se llega al final, se asigna el veredicto esperado (anotación de la última acción).

6. Formalización de la Ejecución de Pruebas

Todos los aspectos comentados han de ser combinados en la formalización de la ejecución de pruebas. Se resalta que existe un tercer elemento en juego que no suele tomar parte en técnicas más convencionales: una especificación de referencia formalizada y manejable por herramientas.

Adoptaremos las siguientes hipótesis de partida:
- La IUT sólo ofrece/ejecuta un evento cada vez.
- La prueba es correcta (según se definió en el punto 5.6).
- Supondremos la existencia de la función ***io***, que determina si una acción de una prueba es imponible o no.

Sea T una prueba correcta, I una implementación, t el tiempo transcurrido desde la ejecución de una acción. Sean t_0 e i_0 los estados iniciales de T e I, respectivamente. Denotaremos con TS, TS', TS_i a conjuntos de estados sobre la prueba, y con O_NIL el evento nulo.

Definición 1. *Definimos como resultado de la ejecución de una prueba sobre una IUT un valor en el siguiente dominio: R = (BloqueoEnPreambulo, BloqueoEnCuerpo, PASA, FALLA, INCONCLUSO).*

Definición 2. *Definimos el estado de una ejecución de una prueba sobre una IUT como la tupla* $\Upsilon = < TS, i >$ donde TS es un conjunto de estados pertenecientes a la prueba e i es un estado de la IUT. Denotaremos como φ, φ_1, φ', ... a elementos de Υ.

Definición 3. *Definimos la variable preamble* $\in$ *{TRUE, FALSE}*. Esta variable tomará el valor *TRUE* si la prueba se encuentra en el preámbulo y *FALSE* en el caso contrario. Operacionalmente, comienza siempre con el valor TRUE y lo cambia cuando se ejecute la acción anotada como END_OF_PREAMBLE.

Definición 4. *Definimos la función de terminación T:offert {PASA, FALLA, INCONCLUSO}* que devuelve la anotación de un evento terminante de una prueba definido en 5.7.

Definición 5. *Definimos la función L: offert {IN, OUT}* que devuelve la direccionalidad de un evento.

Definición 6. *Definimos la función* Γ *que dado un evento devuelve el valor asignado en la anotación* timeout: Γ*: offert* $\rightarrow$ *time*.

Definición 7. *Definimos la función* Ofs *de extracción de eventos dado un conjunto de estados como: Ofs : state** $\rightarrow$ *off_set* con $OFs(TS) = \{o \mid o = Of(t_i) \forall\ t_i\ TS\}$.

Definición 8. *Definimos la función* Ofc *de elección de evento entre una prueba y una IUT como: Ofc :* $\Upsilon \rightarrow$ *offert* siendo:

$$Ofc(<TS_n, i_n>) = O\_NIL \quad si\ t > t_i, \forall\ t_i = \Gamma(o_i) \forall\ o_i\ Ofs(TS_n)$$
$$= Ofs(TS_n) \otimes Of(i_n) si\ L(o_{i_n}) = OUT \wedge t\ leq\ t_i, \quad \forall o_i \in Ofs(TS_n) \forall\ t_i = \Gamma(o_i)$$

EJECUCION	CLASIFICACIÓN				
		MUST ACCEPT	**MAY ACCEPT**	**MUST REJECT**	**MAY REJECT**
	Bloqueo Preámbulo	I	I	I	I
	Bloqueo Cuerpo	F	F	P	P
	Termina	Etiqueta terminal			
	I: Inconcluso ~~ P: Pasa ~~ F: Falla				

Tabla 1: Asignación de veredictos

$$= Ofs(TS_n) \otimes Of(i_n) si\ L(o_{i\,i}) = IN \wedge t\ leq\ t_i,\ \forall o_i \in Ofs(TS_n)\ \forall\ t_i = \Gamma(o_i)$$

En el primer caso, tenemos que si el tiempo transcurrido tras la la última acción ejecutada sobrepasa todos los timeouts especificados se considera que existe un bloqueo. El segundo considera el caso de que, sin existir vencimientos de plazo (al menos en alguna acción observable por la prueba), la IUT impone alguna acción. En definitiva se espera la reacción de la IUT, ahora bien, esa reacción debe ser acorde al propósito de la prueba.

Por último, la prueba intenta imponer una acción a la IUT. Recordemos que en este caso TS_n debe ofrecer una sola acción. Si esta acción no es ejecutada por la IUT, el resultado de $Ofs(TS_n) \otimes Of(i_n)$ será, evidentemente, O_NIL.

Definición 9. *Definimos la función* $A : state^* x\ offert \rightarrow state^*$ $A(TS, a) = \{ s_{i+1} / s_{ni} = a \Rightarrow s_{i+1}\ \forall\ s_i TS \}$ como la función que, dado un conjunto de estados de la prueba y un evento ejecutado, determina el conjunto de estados resultantes en la prueba. Esta función, en definitiva, hace un seguimiento sobre la especificación de los estados de la prueba por los que está pasando la ejecución de la misma.

Definición 10. *Definimos la función* **P** *de paso de ejecución de una prueba sobre una IUT como* $P: \Upsilon\ X\ offert \rightarrow \Upsilon$.
$P(<T, i>, o) = \{ (<T', i'>) / T' = A(T, o), i' = gl(i, o) \}$

Definición 11. *Definimos la función de ejecución de una prueba sobre una IUT como* ξ *: test, IUT* $\rightarrow R$, *donde el comienzo de la ejecución se denota como* $\Upsilon\varphi \in \rightarrow \xi(\{ t_0 \}, i_0)$.

$$\begin{aligned}
\xi(<TS_n, i_n>) &= T(o) && si\ Ofs(TS_n, o) = \varnothing \wedge o \neq O_NIL \\
&= BloqueoEnPreambulo && si\ Ofc(<TS_n, i_n>) = O_NIL \\
& && \wedge\ preamble = TRUE \\
&= BloqueoEnCuerpo && si\ Ofc(<TS_n, i_n>) = O_NIL \\
& && \wedge\ preamble = FALSE \\
&= \xi(P(<TS_n, i_n>), Ofc(TS_n, i_n))) && en\ otro\ caso
\end{aligned}$$

Esta función va evolucionando la prueba acorde con la IUT, decidiendo entre imponer o esperar un evento. Si se detecta un bloqueo, la función determina si ha sido en el preámbulo o en el cuerpo. Una vez terminadas las acciones de la prueba, el resultado es la anotación del evento terminante de la misma.

7. Impacto en parámetros de calidad

En esta sección describen los principales parámetros de calidad y se analiza el impacto de este proceso de formalización:
Corrección: *capacidad de un producto de satisfacer todos los parámetros de usuario.* Todo el proceso de conformidad persigue, precisamente, la *corrección* como objetivo primordial.
Fiabilidad: *capacidad de un producto de llevar a cabo las funciones para las cuales fue diseñado.* Este proceso queda soportado por herramientas automáticas que llevan a cabo la ejecución de pruebas. La reducción del factor humano disminuye la posibilidad de error en la interpretación de resultados de la ejecución de pruebas.
Mantenimiento: *capacidad de un producto para detectar, localizar y corregir errores residuales.* El mantenimiento es facilitado por una Generación Automática de Pruebas y su posterior clasificación según objetivos.
Testabilidad: *capacidad de un producto para ser probado y verificado.* La formalización de la ejecución de pruebas, junto con la norma ISO 9646 suponen un gran incremento en las posibilidades de pruebas de un sistema, tanto por la definición exacta de interfaces como de métodos.
Eficiencia: *capacidad de un producto para usar la cantidad mínima de recursos.* La generalidad de métodos y herramientas puede suponer una disminución en la eficiencia del proceso de pruebas, tanto en los aspectos operativos como de preparación de expertos.
Flexibilidad: *capacidad de un producto para evolucionar.* La existencia de métodos automáticos supone un incremento en la flexibilidad, por la posibilidad de regeneración automática de pruebas y su ejecución.
Reusabilidad: *capacidad de un producto para ser usado en varias aplicaciones con los mismos requisitos funcionales.* Puesto que estos han sido capturados formalmente, sólo la fase de implementación necesitará ser revisada, evitando un rediseño del sistema. Asimismo, la disponibilidad de *Pruebas Abstractas* implica una aplicación directa del sistema ante nuevas implantaciones.
Interoperabilidad: *capacidad de un producto para ser usado conjuntamente con otro(s).* Puesto que es posible demostrar la conformidad del producto respecto de su norma, se garantiza la satisfacción de definiciones de interfaces [DiL91].

8. Conclusiones

En este artículo se aborda la formalización de la fase de ejecución de pruebas de conformidad en un entorno en el que se dispone de una especificación formal del sistema, que consideramos como *referencia*. Se abordan desde aspectos metodológicos *(prueba correcta)* hasta operacionales (ejecución de pruebas), embebido en un entorno normalizado (ISO-9646).

Se define la *prueba correcta*, de crucial importancia a la hora de ejecutar, interpretar resultados y asignar veredictos. En la definición se tienen en cuenta aspectos como el indeterminismo en la especificación de referencia, imponibilidad y observabilidad, y otros que afecta al proceso de ejecución.

Se elabora un modelo *operacional* de ejecución de prueba genérico, no dedicado a ningún producto en particular y orientado a las pruebas de Conformidad.

ISO ha desarrollado una serie de normas (9XXX) con el objetivo de cuantificar la calidad de un producto. Para ello define una serie de parámetros que tienen influencia en la calidad, para posteriormente cuantificarlos en base a ciertas métricas. En esta sección se relata la influencia que tiene la introducción de FDTs en el ciclo de vida, particularizando para la fase de pruebas.

En primer lugar, el simple hecho de utilizar FDTs elimina los problemas de ambigüedades, faltas de precisión que puedan existir en los requisitos de usuario tal como el cliente los describe.

El hecho de que existan Baterías de Pruebas para homologación de productos supone una mejora inmediata en cuanto a *reusabilidad* y *flexibilidad* para determinar la conformidad de un producto. Por otro lado, como se demuestra en el presente artículo, es posible formalizar el proceso de ejecución de pruebas lo cual, junto al hecho de la existencia de los PICS y PIXIT, arroja un resultado positivo

en cuanto al incremento de la *testabilidad (testability)* de un producto.

La propia automatización del proceso, junto a la existencia de métodos automáticos de Generación de Baterías de Pruebas supone una mejora importante en dos aspectos básicos de la calidad del producto: su *corrección*, que queda asegurada, y su *fiabilidad*. Es importante resaltar de nuevo, para no llamarse a engaño, que las FDTs se han pensado con el objetivo de capturar requisitos funcionales. Esto deja de lado aspectos de *robustez*.

Por último, el *mantenimiento* de grandes Baterías de Pruebas es mejorado por el hecho de disponer de herramientas software que las generan automáticamente.

Como aspecto negativo, cabe destacar que las herramientas de manejo de FDTs hoy en día consumen bastantes recursos de memoria y tiempo, con la consiguiente pérdida de *prestaciones*. Afortunadamente, el rápido avance en ordenadores hace que esta faceta se diluya en importancia con la aparición de sistemas más capaces.

9. Referencias

[Dil91] F.Dilonardo. *A comparison between conformance, interoperability performance, development testing.* In CEN/CENELEC & ETSI Editor, Conformance Testing and Certification in Information Technology and Telecommunications, pgs 377-384. IOS Press, 1991. Proc. of the European Conf., Bruselas, 13-15 junio 1990.

[dNH84] Rocco de Nicola and M.Hennessy. *Testing equivalences for Processes.* Theoretical Computer Science, 34:83-133, 1984.

[ISO91] ISO. *Information Processing Systems -Open Systems Interconnection-* Conformance Testing Methodology and Framework. IS 9646, ISO, 1991.

[MdM89] José A.Mañas & Tomás de Miguel. *From LOTOS to C.* In Ken J.Turner, Editor. Formal Description Techniques, I. pgs 79-84, Stirling, Escocia, UK. 1989. IFIP, North Holland. Proceedings FORTE'88, 6-9 septiembre 1988.

[Rob91] Tomás Robles. *Contribución al tratamiento formal de la fase de pruebas del ciclo software en Ingeniería de protocolos.* tesis doctoral, Depto. Ingeniería Telemática, ETSITM, UTM, 1991.

Cartas a Novática

Madrid, 28 de noviembre de 1995

Estimados amigos de NOVATICA:

He recibido ayer la revista NOVATICA de ¡julio y agosto¡ Es sorprendente que cada vez se demoren mas las cartas que recibo desde Barcelona. ¿No sería posible descentralizarlo y enviarlas desde Madrid? Normalmente, las circulares anunciando cursos, etc., que me llegan desde la secretaría de Madrid sí son mas puntuales.

LA REVISTA NOVATICA me ha parecido magnífica número a número. Concretamente éste sobre seguridad informática es apasionante.

POR SUPUESTO QUE EXPRESO MI OPINIÓN A FAVOR DE MANTENER EL ESTILO ACTUAL (con mayor periodicidad y puntualidad), o en todo caso, sumarme a las propuestas que mencionan el incluir calificación del grado de dificultad de los artículos, opinión de los lectores, etc. Pero de ningún modo convertirla en una mas de las revistas comerciales que tanto aundan. Ni incluiría publicidad. El anuncio de Toshiba de la contraportada ya queda bastante decorativo.

Espero que mis comentarios sirvan para mejorar la revista que considero única en España y digna de ser protegida como especie en vías de extinción. Insisto en aumentar su periodicidad y en potenciar asismismo las raquíticas Boletín de ATI y 'Ati Hoja Informativa'. Saludos,

Miguel Angel

Miguel Angel Jiménez Martín;
socio 9073.
Estudiante de Ingeniería Técnica en Informática de Sistemas en la Escuela Universitaria de Informática de la Universidad Politécnica de Madrid.
e-mail: jmartín@seuix.seui.mec.es

Ingeniería del Software

Adaptación de las Inspecciones para su aplicación en PYMES

Fernando Aldea Montero, Isabel Gallego
CRISA - P.T.M.
(aldea@crisa.es)
(igallego@crisa.es)

1. Introducción

Este trabajo presenta un proceso alternativo al propuesto por Fagan para las inspecciones de código fuente en proyectos y empresas no muy grandes, el cual se viene utilizando con éxito en nuestra empresa desde hace aproximadamente 6 años.

El objetivo principal de las inspecciones es detectar el mayor número posible de errores antes de comenzar las pruebas, con el fin de corregir el producto cuanto antes y de mejorar los procesos de desarrollo

Las inspecciones de código no son un proceso aislado en el ciclo de vida del software, sino que se utilizan junto a otras técnicas de verificación estática (revisiones formales, auditorías internas y externas), todas ellas realizadas de acuerdo al plan de calidad de cada proyecto.

2. Proceso de inspección de calidad de código

2.1. Proceso básico

El proceso de inspección, tal y como se interpreta en CRISA, consta de las siguientes fases:

Inspección preliminar: Un ingeniero de calidad revisa el primer módulo libre de errores de compilación de cada programador. El objetivo de esta inspección es detectar y corregir lo antes posible aquellos hábitos del programador que vayan en contra de las normas de estilo establecidas para el proyecto.

Inspección principal: Afecta a todo el código fuente, y consta de dos etapas:

En la primera, es el propio programador quien, con la ayuda de herramientas automáticas, verifica si el código que está produciendo es conforme a los estándares aplicables al proyecto.

Cuando el programador ha subsanado los problemas detectados por las herramientas, entrega el producto a un ingeniero de calidad, el cual realiza una inspección visual por muestreo (utilizando listas de comprobación) de aquellos aspectos que no detectan las herramientas.

Una vez que el código supera la inspección principal, se autoriza el comienzo de las pruebas unitarias.

Inspecciones finales: Pueden ser una o varias. Son realizadas por un ingeniero de calidad al final de pruebas unitarias, de integración o de sistema, y tienen por objeto verificar que la corrección de errores no ha producido una regresión en la calidad, con respecto a la inspección anterior.

En la **figura 1** se muestran los puntos de inspección establecidos durante el ciclo de vida de un producto Software.

2.2. Diferencias con el modelo teórico

El proceso de inspección de código implantado en la empresa es una adaptación del modelo clásico de Fagan. Dicha adaptación está pensada para proyectos no muy grandes (menos de 15.000 líneas de código, y unas 4 ó 5 personas en el equipo de desarrollo). El contexto de trabajo en esta clase de proyectos se caracteriza por lo siguiente:

-Fuerte presión de plazos y costes, lo que impide afrontar la inversión y riesgos iniciales requeridos por el proceso de Fagan.
-Nivel de documentación no muy elevado, lo cual contrasta con las exigencias de formalidad impuestas por las inspecciones clásicas.
-Ausencia de una estrategia coherente de recolección de datos y métricas, lo cual impedirá analizar posteriormente la eficiencia de las inspecciones.
-Cultura de revisión baja, lo cual dificulta la selección de buenos inspectores entre los programadores.

En este contexto, se hacen necesarias algunas simplificaciones del proceso clásico de inspección. Las principales diferencias de nuestra adaptación estriban en lo siguiente:

En cuanto a objetivos:
*Aunque el objetivo principal sigue siendo la detección de errores antes de pruebas, la inspección principal también se plantea como un hito que autoriza el comienzo de las pruebas unitarias.
*Las normas de calidad tienen un mayor peso en las listas de comprobación, debido a varias razones:

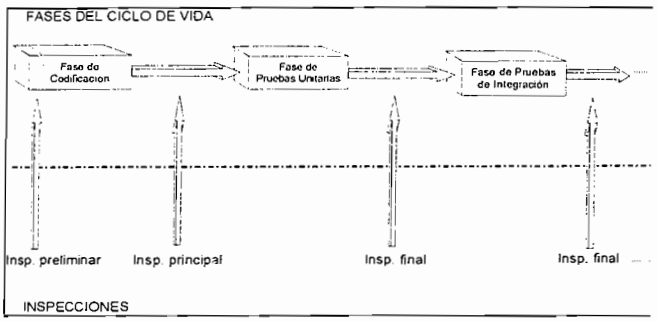


Figura 1: Inspecciones en el ciclo de vida

- No siempre está disponible el diseño detallado actualizado.
- A los programadores les cuesta seguir las normas por sí solos, y a veces necesitan la ayuda de un ingeniero de calidad para interpretarlas correctamente.

En cuanto al equipo de inspección:
*La inspección de código la realizan sólo dos personas: el autor y el inspector (habitualmente, un ingeniero de calidad).
*Hay una etapa en la que el autor realiza una auto-verificación de su propio trabajo.
*A veces el inspector propone soluciones a los defectos encontrados.

En cuanto a las etapas del proceso de inspección:
*No existe la etapa de presentación: el autor se limita a entregar su trabajo al inspector para su revisión.
*Durante la fase de preparación, el inspector revisa el código con ayuda de algunas herramientas y del diseño detallado, y anota los defectos encontrados en una lista.
*No existe reunión de inspección como tal (como máximo se celebra una entrevista entre el inspector y el autor para comentar los defectos encontrados).

En cuanto a la documentación del proceso:
*La lista de defectos contiene los problemas clasificados según su tipo e importancia, y se proporciona información sobre el requisito o norma que se deja de cumplir.
*El informe resumen de los defectos no existe, ya que al haber un solo inspector no tiene sentido.

3. Adaptación de las inspecciones según el lenguaje de programación

El proceso básico descrito en el capítulo anterior debe ser adaptado al lenguaje de programación empleado en cada proyecto. En teoría es posible escribir un programa correcto en cualquier lenguaje, bien sea de alto nivel (Ada, C, Cobol), o incluso de bajo nivel (lenguajes ensambladores). Tradicionalmente se ha comprobado la corrección de un programa básicamente mediante pruebas, utilizando técnicas de caja blanca y caja negra para ejercitarlo a través de un conjunto exhaustivo de casos de prueba. El problema surge por la imposibilidad de verificar el comportamiento completo de un programa de esta manera.

Una solución alternativa es complementar las pruebas con un análisis estático del código fuente, preferiblemente realizado antes de las propias pruebas. En teoría, dicho análisis permite la detección de la mayor parte de los errores del programa, excepto quizás en sistemas de tiempo real.

	C	ADA	C++	COBOL
Abstracción de datos	Baja	Alta	Alta	Nula
Ocultamiento información	Baja	Alta	Alta	Nula
Legibilidad	Baja	Alta	Media	Alta
Tratamiento de errores	NO	SI	SI	NO
Orientación al objeto	Nula	Media	Alta	Nula

Tabla 1

Ciertos lenguajes de programación proporcionan mejores mecanismos que otros para escribir programas consistentes y legibles. Ello contribuye a la fiabilidad global del producto software, y facilita el análisis estático del código. Por el contrario, otros lenguajes permiten (e incluso estimulan) la utilización de trucos extraños que oscurecen el significado del código, lo cual dará lugar normalmente a programas más difíciles de revisar y por lo tanto menos fiables. En la **tabla 1** se resumen las características principales de cuatro lenguajes de uso muy extendido, desde este punto de vista.

4. Listas de comprobación

Debido a estas características propias de cada lenguaje, las listas de comprobación y las herramientas utilizadas en cada caso deben ser distintas. Los principales aspectos incluidos en las listas de comprobación son los siguientes:

* Trazabilidad y completitud con respecto a documentos de diseño.
* Complejidad de subprogramas y módulos, tanto a nivel interno como de interfaces.
* Legibilidad (nivel de comentarios y selección de identificadores auto-explicativos).
* Uniformidad (reglas de nomenclatura, orden de parámetros, misma solución para el mismo problema, etc.).
* Declaración y uso de datos (uso restringido de constantes literales, variables globales, tipos anónimos, etc.).
* Instrucciones del lenguaje (sentencias u operadores cuya utilización está prohibida o restringida, uso adecuado de las expresiones de control, etc.).

5. Utulización de herramientas

La utilización de herramientas en el proceso de inspección puede aliviar el esfuerzo necesario al proporcionar una ayuda para el diagnóstico. No obstante, las herramientas no pueden ser empleadas como sustituto de la inspección visual del código, ya que el actual estado del arte en la medida del software no permite extraer grandes conclusiones de los datos que proporcionan las mismas.

A título de ejemplo, comentamos aquí las características y modo de utilización de las dos herramientas de análisis estático que utilizamos en nuestra empresa:
* *lint* verifica de forma automática ciertos aspectos del lenguaje C normalmente incluidos en los compiladores de otros lenguajes, mediante un análisis estático del código. Entre las comprobaciones que realiza *lint* se encuentran: búsqueda de sentencias inalcanzables, variables no inicializadas, conversiones de tipo peligrosas, uso de sentencias como expresiones, etc. Un problema de esta herramienta es que no existe ninguna especificación suya (de hecho, es un comando estándar de Unix anexo al compilador de C), por lo que es difícil saber qué hace exactamente cada una de las versiones disponibles, con pequeñas diferencias funcionales y de utilización.
**PC-METRIC* es una herramienta simple, pero potente y flexible, que calcula métricas relativas a la complejidad y legibilidad del código fuente. Dichas métricas sirven de

ayuda para determinar cuáles son las partes más críticas del código. Las métricas son recogidas en una serie de informes en formato ASCII, los cuales pueden ser adaptados y analizados según las necesidades de cada proyecto, o incluso exportados a otras herramientas (base de datos, hoja de cálculo).

Por ejemplo, nosotros calculamos la complejidad de un subprograma en función de las siguientes métricas proporcionadas por PC-METRIC: complejidad ciclomática (normal y extendida), nivel máximo de anidamiento de sentencias, volumen de Halstead y número de líneas de código fuente (sin líneas en blanco ni comentarios).

Para cada una de ellas se ha fijado un valor límite, de acuerdo a la siguiente**tabla 2**. De esta manera, construimos una métrica combinada la cual analizamos de acuerdo al siguiente criterio:

- Si un subprograma satisface al menos 3 de los 5 límites anteriores, se considera que su complejidad es aceptable.
- Si viola 3 ó más de los 5 límites, se considera que su complejidad es excesiva y que debe ser diseñado de nuevo (salvo excepción que justifique lo contrario).

6. Conclusiones

Finalmente, se mencionan algunas experiencias útiles obtenidas durante la aplicación de este proceso en nuestra empresa:

* *La inspección preliminar es muy importante*, mucho más de lo que su nombre indica. La corrección a tiempo de aquellos hábitos del programador que sean contrarios a las normas del proyecto ahorrará mucho esfuerzo posterior en inspección y repetición del trabajo.

* *Las herramientas automáticas son sólo una* ***ayuda*** *para las inspecciones*, nunca deben ser utilizadas como elemento decisorio. Por ejemplo, si la complejidad de un subprograma es demasiado alta según PC-METRIC, hay que inspeccionar visualmente el código para ver si está justificado o no (es decir, si el problema que resuelve es intrínsecamente complejo). No hay límites universalmente aceptados para las métricas.

*Durante la inspección principal realizada por el propio programador, éste debe entender cómo deben ser interpretados los resultados de las herramientas. Esto es una consecuencia de lo anterior: el hecho de que un programa no satisfaga los límites establecidos no debe entenderse directamente como que está "rechazado", sino como un aviso ("consultar con calidad").

* *Todos los defectos que se detecten en las inspecciones deben ser corregidos* ***antes*** *de las pruebas unitarias.* Especialmente aquellos que no afectan a la funcionalidad del programa (cuestiones de estilo, legibilidad o estructuración). De lo contrario, el esfuerzo posterior de repetición de las pruebas anulará el ahorro potencial de las inspecciones.

* *Un mal programa debe ser diseñado de nuevo, no explicado.* Si los defectos encontrados en un módulo son importantes, es mejor volver a diseñarlo que intentar arreglarlo con más comentarios u otros cambios estéticos.

* *Ningún proceso o norma pueden ser impuestos a la fuerza a un programador.* Tanto el responsable técnico como el de calidad deben utilizar otros argumentos para convencer a los programadores de la necesidad de su colaboración en el proceso de inspección (razones técnicas, criterios de uniformidad, acción preventiva, etc.).

COMPLEJIDAD DE UN SUBPROGRAMA	
MÉTRICA	**LÍMITE**
Complejidad ciclomática	10
Complejidad ciclomática ,extendida	15
Volumen de Halstead	2500
Máximo anidamiento de setencias	4
Líneas de código fuente	70

Tabla 2

7.Referencias

M. E. Fagan;*Design and code inspections to reduce errors in program development.* IBM System Journal, Vol. 15, Núm 3, 1976.

IEEE-STD-1028-1988; IEEE Standard for Software Reviews and Audits.

Ada Quality and Style, Guidelines for Professional Programmers. Versión 02.00.02. Software Productivity Consortium. 1991.

ESA PSS-05-05; *Guide to the software detailed design and production phase.* Edición 1, Mayo de 1992.

Susan H. Strauss and Robert G. Ebenau;*Software Inspection Process.* McGraw Hill. 1994.

J. Vincent, A. Waters and J. Sinclair; *Software Quality Assurance: Practice and Implementation.* Prentice Hall. 1988.

A.D. Hill; *Elección del Lenguaje de Programación para Software de Alta Fiabilidad.* Una Comparación entre C y Ada. 1991.

Reference Manual for PC-LINT. A diagnostic facility for C. Versión software 5.00. Gimpel Software. Diciembre de 1991.

PC-METRIC User's Guide. Versión 4.00. Set Laboratories.

Ingeniería del Software

Abderraim Bajja
e-mail: 100070.1417@compuserve.com
Fax: 34 3 580 93 64

Tecnologías de la información en España: Análisis y perspectiva

Resumen: *este artículo forma parte de una investigación cuyo objetivo es el estudio, en España, de la práctica de la ingeniería del software en los sectores económicos que utilizan las tecnologías de la información (TI). Pero antes de emprender esta investigación es indispensable averiguar si están disponibles ciertas condiciones básicas. El objeto de este artículo es pues fijar una visión panorámica de las infraestructuras relacionadas con las tecnologías de la información en España. Al intentar situarlas en los contextos europeo e internacional siempre que es posible, tratamos:*
- la política que el gobierno lleva a cabo para fomentar la difusión de las TI.; las telecomunicaciones; los recursos humanos (el potencial, la investigación y la enseñanza) y la producción de hardware y software. El estudio en este artículo no es exhaustivo. Sería muy interesante, para llegar a comprender bien las potencialidades, los puntos fuertes y los puntos débiles de España en materia de TI, establecer una comparación sistemática siguiendo tres direcciones:
- Con los países que tienen potencialidades para convertirse en gigantes de las TI, como India o China;
- Con países cuyo nivel de vida (renta per capita) es similar al de España; por ejemplo, Irlanda o Nueva Zelanda;
- Con países considerados modelos, como Estados Unidos y Japón, sin olvidar Alemania, el Reino Unido, etc.
Es evidente que cada país tiene sus propias particularidades que dependen de sus aptitudes, de sus predisposiciones y de sus realidades económicas. Pero es esencial vigilar la evolución del ambiente de las TI a fin de que todos los actores de éstas se adapten a esta evolución lo mejor posible.

1. Al principio era la información

La información está omnipresente. La difusión de los medios de comunicación (las redes públicas de telefonía y las redes de fibra óptica... para la transmisión de datos) y de la electrónica (ordenadores, módems....) han hecho que los precios de las telecomunicaciones, de los ordenadores y de otros materiales electrónicos relacionados con las tecnologías de la información bajen considerablemente, lo que ha puesto al alcance de una gran parte de la sociedad la obtención, el tratamiento y la transmisión de dicha información.

De hecho, hemos entrado en una etapa compleja y radicalmente diferente de las anteriores. No cabe duda de que vivimos la revolución que preludia la era de la información, en que el valor de base es la información, la tecnología de base es el ordenador interconectado, y el poder económico se concentra en los que pueden crear fácil y eficazmente productos de calidad a buen precio y dejarlos anticuados en seguida, incluso antes de que la competencia reaccione. Aquí, la palabra *crear* debe tomarse en su sentido más general, desde el simple hecho de sintetizar dos o más tipos de informaciones extraídas de Internet y presentarlas con elegancia a un tipo de usuarios, a la creación de nuevos productos o ideas totalmente originales [1, 2] .

La noción de consumidor ha cambiado; la de mercado ha cambiado; la de producto ha cambiado. Y el cambio seguirá afectando profundamente a nuestras sociedades en todas sus dimensiones: social, cultural, económica, e incluso política.

Vivimos una interconexión global. La **tabla 1** muestra claramente cómo se multiplican los *hosts* por todo el mundo, como setas, con tasas de crecimiento importantes. La empresa que quiera participar activamente en la era de la información debe comprender que su mercado potencial dejará de estar limitado por su situación geográfica, que su producto deberá ofrecer valores flexibles capaces de responder tanto a las necesidades de un asiático como a las de un africano, un americano o un europeo... con una condición: la vida de estos productos se ha acortado extremadamente, pues están condenados a quedarse anticuados enseguida.

Varios países se han introducido ya en esta revolución. Mejor aún: ciertos países han encontrado oportunidades en las TI para mejorar sus estrategias de desarrollo económico, ya sea participando directamente en la producción mundial, como es el caso de Singapur, sobre todo en la industria del hardware, ya apoyándose en estas nuevas tecnologías para conseguir ventajas competitivas que les permitan afirmarse, en sus actividades económicas. La **tabla 2** incluye ciertos indicadores de las TI de una muestra de países.

Hay un avance terrible en los diferentes sectores de la tecnología de la información. Japón, Estados Unidos y Europa Occidental se reparten más del 90 % del mercado. Pero países como la India, China y otros tienen también voluntad y posibilidades de participar en este mercado. La India ya ha desarrollado un ordenador, el PARAM, cuyo procesador es paralelo y de un GFLOPS. ¡Y lo ha desarrollado y producido en un período de tres años! [3]. China, con sus 1.178 millones

	enero 94	julio 94	octub. 94	enero 95	aumento 4ºtr.94
América del Norte	1.685.715	2.177.396	2.685.929	3.372.551	26 %
Europa del Este	19.867	27.800	32.951	46.125	40 %
Oriente Medio	6.946	8.871	10.383	13.776	33 %
África	10.951	15.595	21.041	27.130	29 %
Asia	81.355	111.278	127.567	151.773	19 %
Pacífico	113.482	142.353	154.473	192.390	25 %
Europa Occidental	550.933	730.429	850.993	1.039.192	22 %
Total	**2.476.641**	**3.225.177**	**3.898.233**	**4.851.873**	**24 %**

En Europa (enero 95)	
Reino Unido	*241.191*
Alemania	*207.717*
Francia	*93.041*
Italia	*30.697*
España	*28.446*
Resto	*438.100*
Total	***1.039.192***

Tabla 1: Número de hosts de Internet por región mundial (Fuente: Internet Society)

de habitantes, constituye un mercado que ofrece oportunidades colosales. Pero sus 1.172.000 profesionales en informática, con sus bajos salarios y la próxima integración de Hong Kong en China constituirán un importante polo en las TI. Hay otro grupo de países que a pesar de su talla y sus medios modestos reaccionan bien al crecimiento de las TI. Singapur e Irlanda son dos ejemplos claros.

No cabe ninguna duda de que los países que no se toman en serio esta revolución de la informática conocerán la misma suerte que los que no han sabido alcanzar una sintonía con las revoluciones industrial y postindustrial. El ejemplo de las tecnologías orientadas hacia los objetos (TOO) es elocuente para ilustrar en el sector económico, a título anecdótico, lo que desde el punto de vista de la estrategia se juegan las empresas avanzadas en las TI, así como los riesgos que ciertas empresas (y ciertos países) pueden correr si no se aplican en esta revolución. No hay duda de que el paradigma OO está trastornando la manera de producir, e incluso de pensar, el software. Según Peter Wegner [4], las máquinas interactivas, que son extensiones de las máquinas de Turing por agregación de entradas-acciones y que modelizan el paradigma OO, son más potentes que las máquinas de Turing, modelos matemáticos de los paradigmas basados en los procedimientos o funciones. Pero la observación más pertinente es que, según P. Wegner, gracias a esta modelización se puede demostrar que es imposible encontrar especificaciones formales 'completas' de los sistemas orientados hacia los objetos o distribuidos, resultados que resultan fundamentales para los ingenieros de software... 'indisciplinados'. Los partidarios de las tecnologías orientadas hacia los objetos prevén (y ven) que el mundo del software quedará dividido en dos: los dedicados a concebir los objetos y los dedicados a montarlos para responder a ciertas necesidades, de la misma manera que hay países donde se conciben y construyen automóviles y otros donde no se hace más que el montaje (desde luego, con la concentración del valor añadido en los primeros y por lo tanto el dominio del mercado, con todas las ventajas que ello puede conllevar).

2. El caso de España

España es un país con una población de más de 39 millones de habitantes, y con una renta que supera los 13.000 dólares por persona. Su pertenencia a la Unión Europea y su vecindad con países como Francia, Alemania, Italia y Gran Bretaña le confieren una situación confortable en relación a ciertos países en vías de desarrollo como La India, China y otros países del Pacífico, Asia y Europa del Este. Esta situación confortable se caracteriza por el dinamismo de Europa como polo económico y por la importancia del mercado que constituye, tanto en general como, particularmente, en el campo de las TI. Y al formar parte de ese mercado dinámico, España disfruta de posibilidades de mejorar en diversos dominios su competitividad más allá de las fronteras de la Unión Europea. Más aún: España, por su situación estratégica de frontera europea del sur y por su riqueza cultural, tiene posibilidades para desempeñar un papel muy importante en el mundo. No debemos olvidar que el español es la lengua materna de más de 300 millones de personas, una parte importante de las cuales vive en Estados Unidos.

La economía española reciente ha conocido tres fases: Autarquía o período de la dictadura, preapertura, que coincide con el fin de la dictadura y con el período de transición, y finalmente el periodo apertura e integración en la CEE que coincide con la llegada de la democracia. España ha sabido llevar a cabo estas transiciones de una manera ejemplar y envidiable. En los años 80, España conoció un crecimiento sin precedentes en diversos dominios, pero aún le queda mucho camino por delante para consolidar su plaza en el seno de la Comunidad Europea y en el mundo entero. Y ello es particularmente cierto en lo relativo a las tecnologías de la información. En efecto, ciertos acontecimientos incitan a los actores españoles de las TI a reflexionar. Recientemente IBM ha cerrado su sección de desarrollo y adaptación de aplicaciones de software y en junio de 1994 cerró su fábrica de Valencia y transfirió la producción de los sistemas ES/9000, normalmente destinados a la banca y a las grandes empresas, a su fábrica francesa de Montpellier. Por la misma época, Siemens Nixdorff cerró su departamento de desarrollo de software, mientras que en otras regiones del mundo se asiste a cooperaciones (Joint-Venture) como la de IBM y Tata, o la de HP y HCL, y esto acaba de empezar: según The Institut (periódico de IEEE, octubre 1995, vol. 19, nº 10), Informix prevé el desarrollo de su centro en la India en su segundo laboratorio mundial que posee, y también está previsto que el centro de investigación y desarrollo de Novell en Bangalore desempeñe un papel crucial en el desarrollo de SuperNOS, que integrará UnixWare con NetWare.

Como cualquier empresa, las multinacionales (MN) buscan beneficios. Y los movimientos que se observan en ellas pueden interpretarse, simplemente, como estrategias que responden a necesidades específicas como la consolidación en los sectores en que tienen debilidades en relación a la

	Población (millones)	Líneas telefónicas por 1.000 h.	MIPS por 1.000 habitantes	Nº de informáticos por 1.000 h.	Producción HW 1993 (mil. US $)	Gastos en TI (92) como % del PIB
Dinamarca	5,1	577	343	7,52	166	1,5
Finlandia	5,0	542	339	6,94	670	1,1
Hong Kong	5,8	448	159	NA	2.306	1,5
Irlanda	3,5	298	285	6,71	3.729	NA
Israel	4,9	343	171	7,54	464	NA
N. Zelanda	3,4	439	302	7,21	38	2,7
Noruega	4,3	515	357	7,40	335	1,5
Singapur	2,8	365	241	4,11	10.933	2,2
España	39,0	**376	-	1,5	3.175	1,2
Suecia	8,6	690	307	7,51	832	1,3
Estados Unidos	252,5	552	673	7,93	49.380	2,8
Japón	124,0	461	199	7,88	50.939	1,6
China	1.178	12	1	1,0	2100	0,29
India	903	12	1	1,2	476	0,49

*Tabla 2: Indicadores de las TI en ciertos países. Fuente:[5]. Sedisi y elaboración propia. **1994 Cifra facilitada por Telefónica*

competencia, o bien como el deseo de querer estar presentes en los mercados prometedores y en expansión.

Pero cuando se analiza la producción y la balanza comercial del sector informático se observa que las responsables de más del 90 % de la exportación en España son, precisamente, estas MN, y que además constituyen ese 20 % de empresas que participan con el 80 % en toda la producción interna, lo que confirma una vez más la regla de Pareto. Esto nos obliga a buscar en el interior de las fronteras españolas la respuesta a la siguiente pregunta: ¿Cuáles son las razones por las que estas MN disminuyen sus volúmenes de trabajo en España?

Dos palabras clave están de moda: *calidad* y *productividad*. Veremos que las tecnologías de la información han dejado de ser un bien escaso y que el mercado de estas tecnologías ha alcanzado su fase de madurez, en la que el eslogan cambia de *vendemos porque somos los primeros en el mercado* a *vendemos porque somos sofisticados y ofrecemos a los consumidores productos de calidad a muy buen precio.*

Hace apenas dos años, las empresas españolas certificadas ISO9000 eran muy escasas. Actualmente podemos decir que en la industria hay un movimiento hacia la calidad. Pero, ¿cuál es la situación en lo que se refiere a la calidad de las industrias relacionadas con las tecnologías de la información y, sobre todo, en la industria del software?

3. Voluntad política

En España, la toma de conciencia del interés de las tecnologías de la información se remonta a los años 80. El principal agente de la planificación y la puesta en marcha de la política relativa a las TI es el Ministerio de Industria. Los instrumentos utilizados para esta puesta en marcha son los planes nacionales de investigación científica y desarrollo tecnológico y los planes tipo PEIN (Plan Electrónico e Informático Nacional). La mayoría de los programas de estos planes está dirigida a la gestión de ayudas a los proyectos concertados con empresas, cuyo objetivo es la promoción de infraestructuras para potenciar la I+D en las empresas. Constituyen los principales planes de la política pública para las TI a partir de los años 80. La toma de conciencia por parte del gobierno de la necesidad de una política activa data del año 1984 con el PEIN I, que promueve, mediante la utilización de mercados públicos y subvenciones, el establecimiento de multinacionales para reforzar la industria y la transferencia de la tecnología.

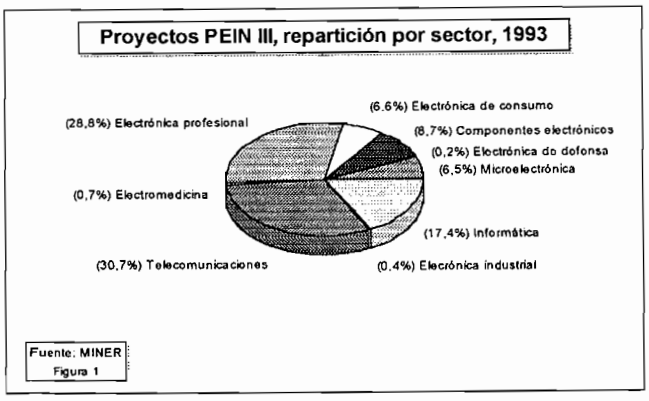
En 1988, con el PEIN II, el Ministerio de Industria quiso reforzar principalmente, mediante su ayuda a las PYME, la alta tecnología en electrónica profesional e industrial y en informática especializada, así como la fusión de empresas de software. La **figura 1** ofrece un ejemplo del reparto de la subvención (3.032,6 millones de pesetas), a través del PEIN III, de ciertos proyectos. La parte dedicada a las telecomunicaciones y a la informática es notable. A partir de su integración en Europa, España participa en los programas comunitarios de investigación y desarrollo precompetitivos ESPRIT (*European Strategic Programme for Research in Information Technology*) y EUREKA (*European Research and Coordination Agency*), cuyos objetivos son promover la cooperación europea transnacional en materia de TI, dotar la industria europea de tecnologías necesarias para afrontar la competitividad de los años 90, y contribuir al desarrollo y a la implementación de los estándares internacionales, así como (EUREKA) suministrar productos y servicios de alta tecnología para preparar a Europa para competir en el mercado mundial.

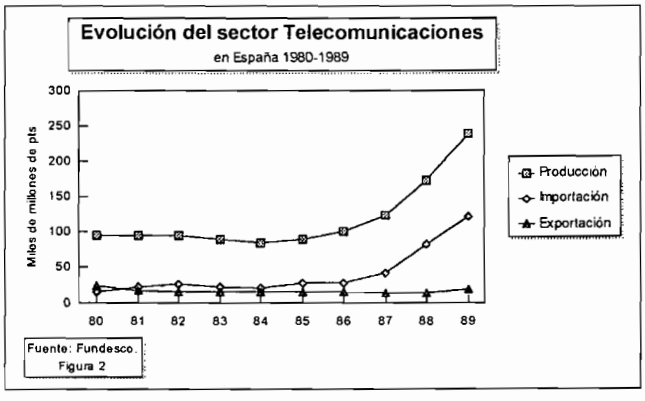
4. Telecomunicación

Las telecomunicaciones constituyen un pilar fundamental en la arquitectura de la era de la información. Y ha sido la convergencia de la informática, la electrónica y las telecomunicaciones la que nos ha permitido alcanzar esta era en que las sociedades avanzan a velocidad electrónica.

Desde principios de los años 80, España, a través de Telefónica, no ha cesado en sus esfuerzos por mejorar su infraestructura de telecomunicaciones. En 1994 existían aproximadamente 376 líneas telefónicas por mil habitantes. Actualmente existen proyectos relativos a la instalación de la comunicación por cable y la telefonía móvil. La **figura 2** muestra la evolución de la producción, la importación y la exportación en el sector de las telecomunicaciones en el curso de la década 80-89, según Fundesco [8]. Se observa una fuerte demanda a partir de 1985.

Desde la apertura de España a la economía internacional y desde su integración en la CEE, las telecomunicaciones se han ido liberalizando poco a poco. Actualmente, varias empresas especializadas en ese sector operan en España; entre ellas podemos citar AT&T, BT Telecommunications, France Télécom, Sprint, Cable & Wireless y Telecom Italia. La mayoría de estas empresas trabaja, de momento, en el mercado de la transmisión de datos, y todas ellas constituyen competidores potenciales temibles para Telefónica. Hemos de observar que el consumidor ha ganado con esta competencia. Los precios han bajado, la calidad ha mejorado y las empresas prestan más atención al cliente y se muestran agradables con él. Un ejemplo elemental, pero a la vez muy profundo, es que se está produciendo un descenso de precios sin precedente para conectarse a Internet. Existen incluso ciertas empresas que permitan al cliente crear y almacenar su propia *Página Web*.





Desgraciadamente, ése no es el caso de la telefonía básica, en la que la única empresa que trabaja es Telefónica. Su facturación supera el billón de pesetas (1.271.510 millones de pesetas), con un beneficio neto de 83.899 millones de pesetas en 1992, según el *Anuario el País 1994*. Al contrario que en el sector de la transmisión de datos, y como en el caso de cualquier monopolio, no podemos evitar pensar que los beneficios se consiguen a expensas del consumidor. En efecto, cuando se produjo el último aumento de las tarifas urbanas, al consumidor sólo le cupo resignarse. Peor aún: parece ser que la métrica que consiste en evaluar el nivel de satisfacción de los consumidores no revela datos positivos. Stephan Doppler escribe en su libro [6]: *Telefónica es una empresa estatal que -todos lo sabemos- funciona bastante mal. Sus servicios telefónicos (de 003, 025, 009 entre otros) son una verdadera pena; su red se derrumba casi todas las mañanas en las grandes capitales[...] en el auricular se pueden escuchar sonidos insólitos al presentarse la supuesta señal de línea libre, sonidos en todas las alturas, con pausas de cualquier duración, ecos a gusto de todos y final súbito [...].*

La **figura 2** revela otro hecho significativo, a saber, que la exportación no ha experimentado el mismo despegue que la producción y la importación, y que se ha quedado casi estancada (la curva, en la figura 2, es casi una recta). ¿Quiere esto decir que en el sector de las telecomunicaciones la única fuente de beneficios es el mercado interior? En todo caso, esta situación no durará mucho tiempo, y las empresas mencionadas están ahí, a la espera, algunas desde 1993, de la liberalización de los servicios de voz o de telefonía móvil que el gobierno llevará a cabo en 1998.

La **tabla 3** revela otro hecho que, sin ninguna duda, no es tan dramático como el anterior: presenta el tráfico de datos de una muestra de países. Resulta interesante ver cómo reaccionan los países pequeños como Irlanda, Singapur y Hong Kong, y los países del Este, como Polonia, para abordar la era de la información. Observemos que en Irlanda, cuya población es diez veces inferior a la de España, el tráfico de datos es casi un tercio del tráfico español.

5. Recursos humanos

La **figura 3** representa la tendencia de la oferta y la demanda en Japón hacia el año 2000. MITI prevé un déficit de casi un millón de profesionales de la informática. Nadie puede poner en duda la voluntad de Japón de participar en la revolución de la información, e incluso de darle forma. Por otra parte, desde hace años Japón está desplegando un gran esfuerzo en las TI. El esfuerzo que hace en I+D en relación con las TI es notable (véase la **tabla 4**). Observemos que en 1985, los gastos de I+D en España eran del 0,55 % del PIB.

Las TI son cada vez más necesarias en los diferentes sectores de la economía de un país. Crean nuevas necesidades y dejan anticuadas las maneras tradicionales de hacer negocios. Por ello, su impacto en el empleo es muy fuerte. En la UE se prevé que se crearán más de tres millones de puestos de trabajo relacionados con los multimedias, y se prevé también que la competitividad de más de 60 millones de puestos de trabajo dependerá fuertemente de los servicios ligados a la información y a las telecomunicaciones. Esto presagia un crecimiento de la demanda de profesionales de las TI, sobre todo los de software. Estamos convencidos de que todo país que intente alcanzar una posición económica de primera clase en la era de la información, esta tendencia debe optar por el crecimiento. Evidentemente, una de las razones de la depresión de la curva que representa el empleo en el sector de la informática en España (**figura 4**), es la recesión que el país vivió en 1993, año en que el PIB bajó a **-1**. Pero no es ni mucho menos la única razón. Mientras en los países donde las TI siguen una evolución normal, de cada 1.000 personas más de 6 trabajan en el sector de la informática, en España apenas 1,5 están en dicho sector. Si a esto añadimos que un hindú produce más de 60 veces su salario mientras que un español produce algo más que 5 veces su salario, y que la estructura de la industria de la informática nacional es débil (formada por pequeñas empresas que carecen de los medios materiales y tecnológicos necesarios para plantar cara a las multinacionales, y que operan en un medio desarticulado), sólo se puede concluir que el déficit de profesionales de software, actualmente estimado en más de 200.000 personas, será muy difícil de resolver en los próximos años.

La enseñanza

La informática es una actividad intelectual, y escribir programas no está al alcance de todo el mundo, pues es necesario un mínimo de abstracción. La enseñanza (así como la formación) desempeña y desempeñará un papel fundamental para establecer sólidamente las TI en general y para formar personas *disciplinadas* en ingeniería del software en particular. La **tabla 5** proporciona estimaciones de los usuarios finales que programan en una muestra de países. No exageramos al decir que en el futuro el alfabetismo se medirá por el número de personas que sepan leer, escribir y, *además*, interactuar con un ordenador personal.

Según un estudio realizado por Fundesco en 1992 [9], hay más de 141 centros en España que ofrecen enseñanzas relacionadas con las TI, entre los que se cuentan grandes universidades cuyo número de estudiantes supera los 30.000. Este estudio indica que la oferta de nuevos diplomados en TI ha pasado de 3.544 en 1986 a 6.607 en 1992. El número de diplomados en software ha pasado de 960 en 1986 a 2.464 en 1992. En el mejor de los casos, para cubrir el déficit anteriormente mencionado harían falta más de cinco años con un crecimiento del 100 %, lo que supone una política agresiva y de choque.

Pero las cosas no son tan ideales como se desea. Hace apenas diez años, la enseñanza de la informática estaba aquejada de diversos problemas en diversos países: el personal no era suficiente, y el material, que suponía un presupuesto importante, tampoco. La confusión era total: no se sabía qué departamento debía encargarse de ella. ¿Sería el departamento de electrónica, el departamento de matemáticas o un departamento de informática? El debate sobre la enseñanza de

Alemania	G.Bretaña	Francia	Holanda
329,83	328,44	310,61	252,96
Italia	**España**	**Irlanda**	**Polonia**
67,21	46,36	13,90	31,39
Hong Kong	**Singapur**	**Japón**	**EEUU**
65,92	92,60	280,72	25.244,18

Tabla 3: Tráfico de datos Ene. 1995 (GigaBytes). Fuente: Internet Society / propia

	CEE	EEUU	Japón
Gasto en I+D *(billones ecus)*	70	123	42
% de I+D frente al PIB	2,3	2,7	2,6
I+D en relación con TI *(% del gasto total)*	19	27	34

Tabla 4: Gastos en I+D en 1984. Fuente: Eurostat/MITI/BAH.

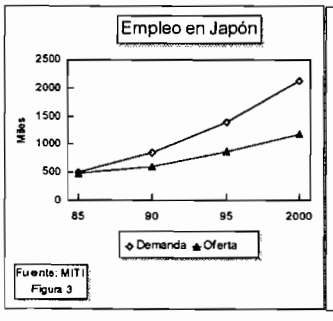
la informática sigue abierto [11]. Ello se debe, en gran parte, a la juventud de la materia que se desarrolla vertiginosamente.

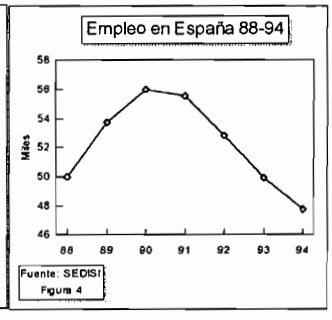
La juventud de la informática ha creado una gran confusión en los espíritus de diversas personas de diferentes categorías que están relacionadas con ella. El artículo de Jorge Palacios *Demanda del sector bancario [9]* constituye un ejemplo brillante. En dicho artículo, tras reprochar a la nueva generación de informáticos que no sepan hablar y que no sepan programar en Cobol, se lee: *Soy un ingeniero del plan 57, [...] he dedicado largas horas al estudio de los espacios de Hilbert de orden n, [...], he de confesar que lo que más útil me ha sido en toda mi vida profesional [...], es la programación de Fortran [...]* (*sic*). Este párrafo, sentimos decirlo, carece de perspicacia. Pauli diría que ni siquiera es falso. La historia de las ciencias, e incluso la de la Humanidad, siempre ha mostrado que las matemáticas eran y son útiles. A la *pregunta ¿Qué consejo daría a un joven ingeniero que este empezando su carrera profesional?*, Knuth respondía: *Practique la escritura técnica tanto como pueda. Lea materiales originales escritos por científicos, matemáticos e ingenieros importantes de los últimos 1.000 años. Mejore lo que no haga del todo bien para que sus puntos débiles se conviertan en fuertes*. Otro párrafo revelador en el mismo artículo es el que describe las necesidades *simples y vulgares* (*sic*) de la banca en particular. El problema que propone como ejemplo es convertir 1.200 programas de Ensamblador a C. La dificultad de ese problema está lejos de ser cuestión de modas *aprender C* frente a *aprender Ensamblador*. Está en el meollo de la ingeniería del software. No es sólo un problema de enseñanza, sino de los bancos, de las empresas, resumiendo: de toda la comunidad que mantiene una relación más o menos estrecha con la informática. Para no entrar en detalles, recordemos que la Reingeniería del Software y la Ingeniería Inversa tratan ese tipo de problemas.

No cabe duda de que la enseñanza tiene sus propios problemas. Uno de ellos es la colaboración insuficiente de la industria. En efecto, contrariamente a lo que pasa en otros países como Francia y Alemania, donde la práctica, el *stage* es una asignatura completa que puede durar de dos a ocho meses, en España, dicha *asignatura* no figura en los programas de enseñanza de la informática, y puede que tampoco en los de ingeniería.

Investigación e industria

Para fomentar una industria informática, un país debe crear una sinergia entre la enseñanza, la investigación y la industria. La investigación científica y el desarrollo tecnológico desempeñan un papel esencial en los países industrializados, pues la innovación y la creación figuran entre las claves de la competitividad. La **tabla 6** muestra los gastos de ciertos países de la OCDE en materia de investigación. España, joven democracia, ha realizado un enorme esfuerzo: de 0,55% en 1985, sus gastos en I+D pasaron al 0,87 % del PIB en 1992.





Desgraciadamente, aunque dignos de alabanza, estos esfuerzos siguen siendo insuficientes y están aún lejos de la media europea. Ello contribuye a aumentar los problemas que vive la investigación en España. De un estudio publicado por Fundesco [12] relativo a los grupos de investigación en TI se desprende que no hay ni un solo medio propicio a la investigación en TI. A continuación ofrecemos un resumen de las conclusiones a que han llegado los autores de dicho estudio; la tabla ofrece las estadísticas:

- No hay investigación con dedicación exclusiva: los investigadores son de hecho profesores investigadores.
- Personal insuficiente. En particular, el personal administrativo es casi inexistente.
- Finanzas y gestión: diversos organismos, entre ellos CAICYT y CSIC, gestionan la concesión de ayudas, lo que implica una burocracia excesiva.
- Infraestructura y medios materiales insuficientes.
- Falta de relaciones intensas con la empresa: da poco valor al título de doctor (no se interesa por temas que no den resultados rentables a corto plazo).
- Problemas de coordinación entre los grupos.

No cabe duda de que existen signos de mejoría. Pero, ¿son suficientes estas mejorías para confirmar que España está en una vía sólida para enfrentarse a la era de la información en una posición de economía de primera clase?

6. Organizaciones relacionadas con TI en España

Las asociaciones son muy importantes para promocionar las TI mediante intercambios de informaciones. Sigue una muestra representativa de varios tipos de organismos y asociaciones que trabajan en el campo de las tecnologías de la información.

ATI (Asociación de Técnicos de Informática): mantiene una relación estrecha con ACM. Entre sus actividades, organiza cursos de reciclaje o de formación en los diferentes temas que están relacionados con las TI. Colabora en la difusión de las TI patrocinando congresos y conferencias. Publica un boletín mensual y una revista bimestral, *Novática*. Esta asociación, que cuenta aproximadamente con 5.000 socios, refleja problemas similares a los que las TI viven en España: falta de personal voluntario, falta de *refrees* para la revista... **(tabla 8)**

ESI-Bilbao European Software Institute: al contrario que el SEI (Software Engeneering Institute, EEUU), pocas empresas españolas conocen su existencia. Algunas tímidas apariciones aparte, por lo que sabemos desempeña el mismo papel de representación que ciertas empresas para determinadas MN.

SEDISI [9] Sociedad Española de Empresas de Informática, representa a 140 empresas que ocupan al 50% del personal del sector y facturan 560.000 millones de pts. **ANIEL** [9] Asociación Nacional de Industrias Electrónicas, representa a 112 empresas.

País	Profesionales SW	Usuarios finales programadores
Estados Unidos	1.175.000	10.000.000
Japón	850.000	3.500.000
Reino Unido	385.000	3.500.000
Francia	375.000	1. 700.000
Alemania	550.000	1.650.000
India	750.000	1.200.000
China	950.000	1.250.000
España**	47.743	160.000

Tabla 5: Estimación de los usuarios finales programadores. Fuente: C. Jones [10].
***Fuente: Sedisi. * Estimaciones con un margen de error considerable*

	Alem.	Franc	Italia	G.Br.	Españ	CE	EEUU	Japón
91	2,65	2,42	1,32	2,13	0,87	1,97	2,67	2,87
92	2,65	2,36	1,38	2,12	0,87	1,94	2,68	2,80

Tabla 6: Gastos Totales de I+D sobre PIB. Fuente: OCDE.

7. Producción de hardware y software

Desde los años 80, el gobierno español no ha cesado de fomentar la difusión de la tecnología de la información, y desde la integración de España en la CEE, ello no ha hecho sino aumentar. En España, las TI han dejado de ser un bien escaso. Se estima que actualmente existen más de 55 ordenadores personales por cada 1.000 habitantes, cifra importante si se compara con la de China o la de la India, donde sólo hay un ordenador personal por cada 1.000 habitantes (a título de referencia: Estados Unidos: 287 ordenadores personales por 1.000 habitantes; Singapur: 125 ordenadores personales por 1.000 habitantes; Japón: 97 ordenadores personales por 1.000 habitantes, en 1993).

La **figura 6** (nota del editor: no existe figura 5) muestra la evolución del mercado neto en los años 80-94. Pone de manifiesto un hecho importante, a saber, que el mercado está al principio de su fase de madurez. Hemos de subrayar que en la década de los 80, la media de crecimiento fue de ¡23 %! ¡Una cifra colosal! Aunque en 1993 este mercado sufrió una baja a causa de la recesión, se espera una recuperación de la demanda (con, esta vez, una tasa racional de crecimiento característica de los mercados maduros). Ello se debe a la subinformatización de ciertos sectores y a la importancia de las TI como herramientas decisivas para la competitividad, tanto en el mercado interior español como en los mercados europeo e internacional.

Los españoles gastan actualmente algo más del 1,2 % de su PIB en las TI. Es evidente que esta cifra constituye un esfuerzo muy grande para informatizarse. Pero continúa estando por debajo de la media de la Unión Europea y siendo inferior a la de ciertos países comprometidos con las TI, como Singapur, donde es del 2,2 % (véase **tabla 2**).

La facturación total bruta en TI (hardware, software y servicios) ha experimentado un crecimiento del 9,2 % y ha alcanzado la suma de 933.383 millones de pesetas, de los cuales 143.123 corresponden a las exportaciones [7] (véase en la **figura 7** una representación gráfica). A pesar de esto, el mercado interior bruto sigue siendo modesto en comparación con los de los países motores de la Unión Europea (véase la **figura 8**). Un hecho notable es la producción de hardware en Irlanda, que alcanzó el valor de 3.729 millones de dólares en 1993. La estrategia de Irlanda se considera un éxito de la *globalización a partir de la nada* [13]. Dicho éxito se debe a que diversas multinacionales localizan sus productos en este país, que

Dominio	**técnicos**	**doc-tores**	**técnic. super.**	**TOTAL**	***proyect. en curso***
Arquitectura/tecnol. de ordenadores	188	39	72	**299**	*27*
Automática	347	76	111	**534**	*54*
Inteligencia artificial	159	39	69	**267**	*16*
Micro/Optoelectrónica	229	82	76	**387**	*28*
Radiocomunicación	186	62	44	**292**	*27*
Redes/Serv.telemátic.	125	15	44	**184**	*19*
Informática	111	26	47	**184**	*18*
Transductores	64	29	21	**114**	*31*
Tratamiento de señales	169	46	60	**275**	*52*
Total	**1.578**	**414**	**544**	**2.536**	***272***

Tabla 7: Grupos de investigación en las TI en España. Fuente: Fundesco-CSIC, 1987

	EEUU	España	Irlanda	mundo
Nºsocios al 11-08-95	59.970	1.176	179	**93.976**

Tabla 8 Nº de Socios de IEEE. Fuente IEEE.

forma parte de la Unión Europea, que cuenta con una mano de obra preparada y no excesivamente cara, donde la lengua nativa es el inglés, y cuya política gubernamental se basa en facilitar el establecimiento de MN mediante variados estímulos. La figura 9 muestra la evolución en España del parque de ordenadores personales, cuyas 2,135 millones de unidades en 1994 pueden compararse con los 928 sistemas grandes, 4.745 medios y 74.368 pequeños.

Software

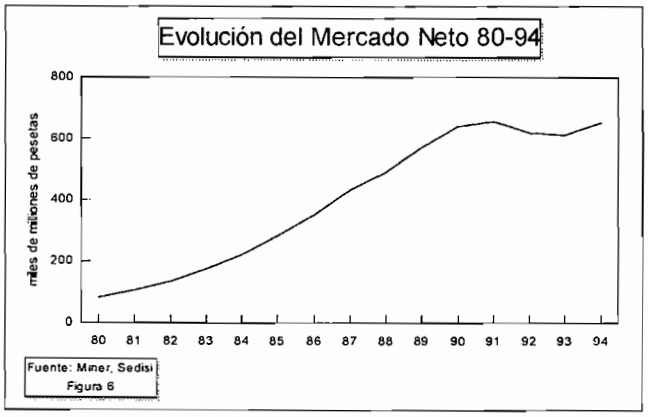
Ciertos países en vías de desarrollo consideran que la industria del software es un medio estratégico para fortalecer sus economías. Hungría, la India, Israel y Chile constituyen ejemplos interesantes. En efecto, los productos de software son esencialmente de puro valor añadido. Al contrario que otras industrias, la del software necesita una inversión inicial mínima.

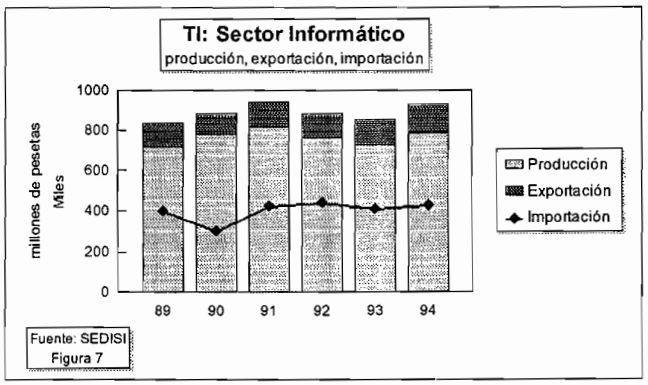
Las componentes servicios y software del mercado interior bruto español han experimentado respectivamente crecimientos del 4,8 % y el 3,0 %, y han alcanzado los 188.981 y 84.364 millones de pesetas [7]. Para comprender el significado de estas cifras, las situamos en el contexto internacional.

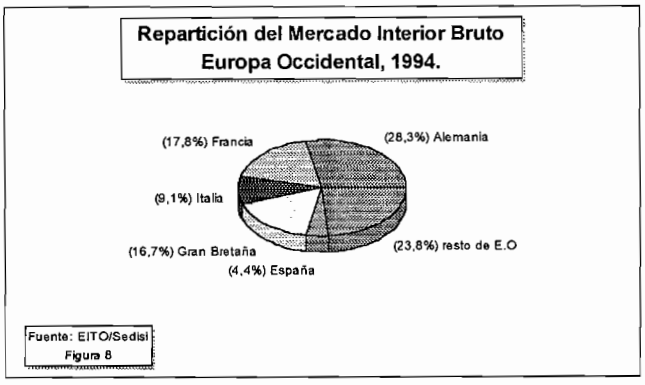
En primer lugar hay que tener en cuenta que Sedisi define el software como el conjunto de productos que permiten *el funcionamiento lógico del hardware (sistemas operativos, herramientas de desarrollo, bases de datos y programas de comunicaciones) y también las aplicaciones estándar que se comercializan bajo forma empaquetada [...] (sic) [7].* No incluye el software a medida. Esto significa que una parte muy grande de esta producción no se realiza en España.

Dado que el software a medida se considera un servicio, analizamos las exportaciones de las dos componentes, software y servicios, como un todo. La exportación de servicios no alcanza más allá de los 5.767 millones de pesetas, lo que representa el 0,08 % del mercado interior bruto de la Europa Occidental, que es de 6,84 billones de pesetas si se excluye la parte correspondiente a España. Esto muestra que la componente servicios es poco competitiva en Europa.

Las exportaciones de servicios y de software alcanzaron la cantidad de 24.066 millones de pesetas en 1994. En 1989, la cantidad alcanzada fue de 22.602 millones de pesetas, lo que







representa, en el mercado mundial, una participación del 0,51 %, cifra que resulta insuficiente para defender una industria del software seria.

Pero lo que aún resulta más alarmante, y debe pues hacer reflexionar a los actores de las TI (estrategas, ejecutores, industria, administración pública, profesionales, estudiantes, etc.), es que la competitividad está cayendo de forma dramática. Mientras el mercado mundial se ha multiplicado por dos, las exportaciones españolas de software permanecen estancadas (véase la **tabla 9**). ¡Del 0,51 %, la participación ha bajado al valor de 0,24 %!

8. Conclusión

España ha dado grandes pasos hacia delante en el campo de la tecnología de la información. La democratización del país y su integración en la Comunidad Europea han desempeñado un papel decisivo en estos avances. Actualmente existen grandes oportunidades para que el papel de España en dichos dominios sea esencial. El país tiene posibilidades y puede tener medios. Es necesaria, y tal vez vital, una *política* de convergencia de los diferentes actores de las TI basada en la efectividad. El ejemplo de Singapur, con la formación de 10.000 profesionales en software para implantar las TI, la intervención del estado en la estrategia de puesta en marcha de las TI en Irlanda, las prácticas en la formación de futuros profesionales en software en Francia, y otras políticas son efectivas y pueden adaptarse a las especificidades españolas.

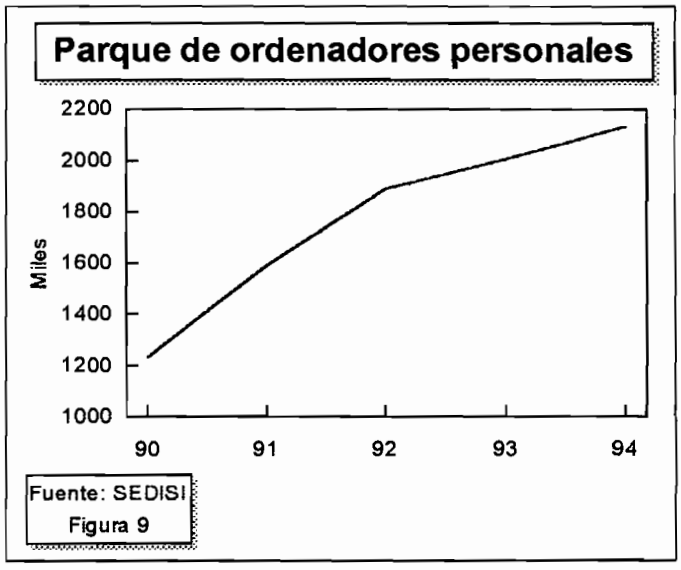
Si España no reacciona seriamente en estos campos, correrá el riesgo de limitarse a proporcionar a Europa y a la comunidad internacional ni más ni menos que buenos consumidores disciplinados.

9. Bibliografía

Lewis, T.; *Living in real time, side A*, Computer, vol. 28, Nº 9, September 1995.
Lewis, T.; *Living in real time, side B*, Computer, vol. 28, Nº 10, October 1995.
Nidumolu, S.E. y Goodman, S.E.; *Computing in India: An Asian Elephant Learning to Dance*, /global_net @ dvhx20.csudh.edu (155.135.1.1).
Wegner, P.; *Interactive Foundations of Object-Based Programming*, Computer, vol. 28, Nº 1, October 1995.
Dedrick y al.; *Little Engine that Could: Computing in Small Energetic Countries*, CACM, vol. 38, Nº 5, May, 1995.
Doppler, S.; *Ganar Dinero y Hacer Carrera*, Ediciones Delfín, 1995.
Sedisi, *Sector Informático y Parque de Ordenadores en España*, Miner, Sedisi, 1994.
Fundesco; *Ciencia, tecnología e industria en España*, Ed. R. Dorado et al., Fundesco, 1991.
Fundesco; *La formación universitaria en tecnología de la información*, M. Gamella et al., Fundesco, 1992.
Jones, C.; *End-user Programming*, Computer, vol. 28, Nº 9, September, 1995.
Gal-Ezer y al.; *A High School Program in Computer Science*, Computer, vol. 28, Nº1, October 1995.
Fundesco; *Comunidad Científica española en las Tecnologías de la Información*, Fundesco-CSIC, 1987.
Grimes, S.; *Information Technology and the Periphery: The Case of Ireland*, Info. Tech. Develop. Count. vol. 4, Nº 1, Jan. 1994.

Software	1989	1994
Total del mercado mundial	36.733	77.492
Exportaciones españolas	190	188
Participación española en el mercado mundial	0,51 %	0,24 %
1989: 1 $ = 119 pts; 1994: 1 $ = 128 pts		

Tabla 9: Millones de $ EEUU



Sistemas abiertos

Jordi Vintró
Tecsidel, Barcelona

Uso de POSIX Threads en Programación OO

1. Threads y programación OO

1.1. Objetos y comportamientos asíncronos

Las aplicaciones reales están a menudo afectadas por un comportamiento asíncrono. Esto ocurre así para la mayoría de aplicaciones de tiempo real, siempre que exista más de elemento externo que genere señales que deban ser atendidas simultáneamente. En dichas circunstancias se hace necesario algún tipo de multiprogramación. Y es aquí donde el uso de un sistema operativo de tiempo real es interesante.

Las tareas simultáneas, sin embargo, se supone que pertenecen todas a la misma aplicación. Esto significa que tienen algunos elementos compartidos, y más precisamente, algunos datos comunes y compartidos. En terminología de objetos, puede decirse que se trata de objetos entidad que son compartidos por diversas tareas. Estos objetos pueden estar ubicados tanto en memoria como en disco.

Si se utiliza un sistema operativo como LynxOS, las tareas pueden implementarse por medio de procesos Unix o bien por medio de 'threads' en el interior de un proceso único. La opción más conveniente depende de la naturaleza de la aplicación y de los requerimientos de persistencia de los objetos.

1.2. Implementación de tareas como procesos

Las tareas pueden ser implementadas por procesos Unix montados (linked) de forma separada.

Si los objetos deben ser persistentes, es necesario guardar los datos en disco. Por lo que respecta a los objetos compartidos, el disco puede ser usado como una especie de memoria común: los datos se guardan solamente en disco y los procesos acceden al mismo cada vez que lo necesitan.

Una buena solución consiste en usar una base de datos orientada a objetos, o incluso simplemente una base de datos relacional. La opción de bloqueo disponible en cualquier base de datos relacional puede usarse, por ejemplo, para asegurar la integridad cuando diversos procesos pueden modificar los mismos datos al mismo tiempo: todos excepto uno de ellos se quedan en espera. Cada objeto entidad se concibe, en este caso, como una serie de registros (una tabla en términos de base de datos relacional). En cada proceso que deba acceder a la misma, debe incluirse una instancia de este objeto. La multiplicidad de instancias del mismo objeto no es un inconveniente, ya que puesto que los datos compartidos se encuentran en la base de datos, no hay datos compartidos en los atributos del objeto.

Si no fuera necesario asegurar la persistencia de los datos y, sobre todo, en el caso de que existan restricciones de tiempo que desaconsejen o imposibiliten el uso del disco, entonces la implementación de la aplicación como un conjunto de procesos resulta más complicada.

En este caso se hace necesario instalar el objeto entidad propiamente dicho en uno de los procesos, que asume el papel de servidor, e instalar 'proxys' en los demás procesos, que actúan como clientes. Cada 'proxy' efectúa llamadas al servidor por medio de primitivas de comunicación inter procesos. El servidor accede al disco y devuelve los datos.

De forma alternativa, los datos compartidos pueden guardarse en memoria común. La mayoría de sistemas operativos proporcionan mecanismos que permiten que diversos procesos compartan datos en memoria. En cualquier caso, se debe prever algún mecanismo de protección (un semáforo, p. ej.) para evitar escrituras simultáneas de los mismos datos.

1.3. Principios de programación OO con threads

La ventaja de usar 'threads' es que las mismas se montan (link) todas juntas, y por tanto es innecesario definir explícitamente una zona de memoria compartida puesto que toda la memoria del proceso se comparte entre todas las 'threads'.

Los objetos entidad pueden implementarse como objetos simples. Sus atributos privados pueden contener datos fácilmente accesibles por todas las 'threads' simplemente declarándolos como públicos. No hay pues entonces ninguna necesidad de replicar objetos ni de de usar 'proxys'.

Naturalmente, diversas 'threads' pueden necesitar acceder simultáneamente a los mismos datos. Puede incluso suceder que un mismo método de acceso sea llamado al mismo tiempo por diversas 'threads'. Dicho acceso puede resultar peligroso según cual sea la estructura de los datos. Si una 'thread' ha leído datos para modificarlos, puede no ser conveniente modificar entonces dichos datos desde otra 'thread'.

Así pues es necesario crear un mecanismo de bloqueo. El 'mutex' que proporciona LynxOS es particularmente adaptado para este fin. Su razón de ser es precisamente la protección de zonas de memoria por diversos accesos en paralelo.

El 'mutex' debería también usarse al leer datos si los mismos deben ser consistentes. De esta forma se asegura que nadie puede modificar los datos mientras son leídos.

Normalmente se crea un único 'mutex' en el interior de cada objeto entidad. Todas las secuencias críticas empiezan reservando el 'mutex' y terminan liberándolo. Así se asegura el acceso exclusivo a los datos para todas las 'threads'. Lo interesante es que todo este mecanismo está encapsulado en el objeto entidad y los demás objetos no necesitan conocer nada de ello.

2. Una propuesta de encapsulación de threads

Los 'mutex' y las variables de condición se pueden encapsular fácilmente como objetos C++. Un 'mutex' se usa habitualmente para evitar el acceso simultáneo de diversas 'threads' a una zona de datos.

En la **figura 1** que sigue, el 'mutex' se representa según las convenciones de Objectory. Las relaciones que se muestran son del tipo 'consistsOf' ('consisteEn'). Esto significa que el 'mutex', así como los datos de tipo privado, se encapsulan en el objeto propietario 'owner' como atributos locales.

El código que sigue muestra la definición y la implementación de la clase Mutex. Una instancia de dicha clase define un 'mutex' LynxOS. Puede ser declarado donde se necesite aunque típicamente se definirá como un atributo de una entidad objeto. En este caso se puede usar usar para proteger otros datos privados. El código que sigue ha sido desarrollado y probado en LynxOS v2.2.1.

```
// Mutex.h
// Mutual exclusion.
//-----------------------------------------------------------
#ifndef _Mutex
#define _Mutex
#include <pthread.h>
class Mutex {
public:
  Mutex (void);              // Constructor
  void lock (void);          // Mutex locking
  void unlock (void);        // Mutex unlocking
  int trylock (void);        // Mutex locking, only if it was unlocked
                             // return 0: it was already locked
                             // return 1: it was not locked; it is now
 pthread_mutex_t sys_mutex;// Operating system mutex; it is defined
                             // as public only to be used by class
                             // Condition_var
};
#endif

// Mutex.cc
//    Mutual exclusion.
//-----------------------------------------------------------
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
#include «Mutex.h»

// Constructor
//-----------------------------------------------------------
Mutex::Mutex (void)
{
  pthread_mutexattr_t mutexattr;    // Mutex attributes
  pthread_mutexattr_create (&mutexattr);
  if (pthread_mutex_init (&sys_mutex, mutexattr) == -1) {
     perror («pthread_mutex_init»);
     exit (-1);
  }
}

// Mutex locking
//-----------------------------------------------------------
void Mutex::lock (void)
{
  if (pthread_mutex_lock (&sys_mutex) == -1) {
     perror («pthread_mutex_lock»);
    exit (-1);
  }
}

// Mutex unlocking
//-----------------------------------------------------------
void Mutex::unlock (void)
{
  if (pthread_mutex_unlock (&sys_mutex) == -1) {
     perror («pthread_mutex_unlock»);
    exit (-1);
  }
}
// Mutex locking only if it was unlockedrn 0: ja estava reservat
//    Return 0: it was already locked.
//    Return 1; in was unlocked. Now it is locked.
//-----------------------------------------------------------
int Mutex::trylock (void)
{
  if (pthread_mutex_trylock (&sys_mutex) == -1) {
    if (errno != EINTR) {
       perror («pthread_mutex_lock»);
       exit (-1);
    } else {
       return 0;
    }
  } else {
    return 1;
  }
}
```

Las variables de condición (**figura 2**) se definen también en la parte privada de los objetos, en general objetos thread. Una variable de condición incluye su mutex asociado.

El código que sigue muestra la definición e implementación de la clase Condition_var. Una instancia de dicha clase define una variable de condición y su 'mutex' asociado. Una tal instancia deberá definirse para cada 'thread' que deba sincronizarse con otra 'thread'.

```
// Condition_var.h
// Condition variable. Each condition variable has an associated mutex.
//-----------------------------------------------------------
#ifndef _Condition_var
#define _Condition_var
#include «Mutex.h»
class Condition_var
{
public:
  Condition_var (void);      // Constructor
  void lock (void)           // Associated mutex locking
    {mutex.lock ();};
  void unlock (void)         // Associated mutex unlocking
    {mutex.unlock ();};
  int trylock (void)         // Associated mutex locking, only if it
                             // is not locked.
                             // return 0: it was already locked
                             // return 1: it was not locked; now it is
    {return mutex.trylock ();};
  int wait (                 // Wait for variable condition;
                             // return 0: time expiration
                             // return 1: variable activated
    int timeout = 0);        // Timeout in milliseconds;
                             // if 0, indefinite waiting.
  void signal (void);        // Condition variable activation
```

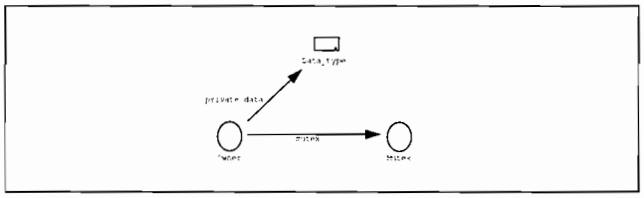


Figura 1

```
private:
   pthread_cond_t sys_cond;// Operating system candition variable
  Mutex mutex;                  // Associated mutex
};
#endif
```

```
// Condition_var.cc
//    Condition variable.
//-----------------------------------------------------------
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
#include <timers.h>
#include «Mutex.h»
#include «Condition_var.h»
```

```
// Constructor
//-----------------------------------------------------------
Condition_var::Condition_var (void)
{
  pthread_condattr_t condattr;      // Condition variable attributes
   pthread_condattr_create (&condattr);
   if (pthread_cond_init (&sys_cond, condattr) == -1) {
      perror («pthread_cond_init»);
      exit (-1);
  }
}
```

```
// Wait for condition variable. Return 0: timeout expiration.
//                          Return 1: condition variable activated.
//-----------------------------------------------------------
int Condition_var::wait (
  int p_timeout = 0)                // Timeout in milliseconds
                                    // if = 0, indefinite waiting
{
  timespec timeout;                 // timeout

                                    // indefinite waiting
  if (p_timeout == 0) {
     if (pthread_cond_wait (&sys_cond, &mutex.sys_mutex) == -1) {
        perror («pthread_cond_wait»);
        exit (-1);
     } else {
       return 1;
     }
                                    // timed waiting
  } else {
     timeout.tv_sec  = p_timeout / 1000;
     timeout.tv_nsec = (p_timeout % 1000) * 1000;
   if (pthread_cond_timedwait (&sys_cond, &mutex.sys_mutex,
                                                    &timeout)
         == -1) {
        if (errno != EINTR) {
           perror («pthread_cond_wait»);
           exit (-1);
        } else {
           return 0;
        }
     } else {
       return 1;
     }
  }
}
```

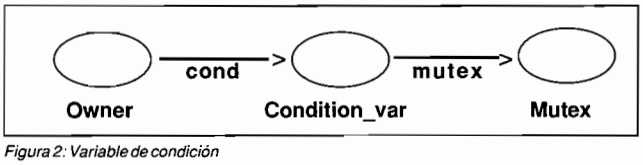


Figura 2: Variable de condición

```
// Activate condition variable
//
//-----------------------------------------------------------
void Condition_var::signal (void)
{
   if (pthread_cond_signal (&sys_cond) == -1) {
      perror («pthread_cond_signal»);
      exit (-1);
   }
}
```

La encapsulación de 'threads' es algo más delicada. Cuando se crea una 'thread', la dirección del código correspondiente debe indicarse en la llamada. Sin embargo, el código no está ligado a ninguna instancia.

En programación orientada a objetos es conveniente ligar todas las rutinas a instancias, como es el caso de los métodos. La forma de solventar este aspecto consiste en hacer que todos los objetos que deban arrancar 'threads' hereden de la clase Thread. De esta forma, cuando se ejecuta el constructor de la clase, un cierto número de 'threads' arrancarán (el número de 'threads' a arrancar se pasan al constructor en un parámetro) **(Figura 3)**.

Esto se muestra en el código que sigue. La rutina de arranque es la misma para todas las 'threads' ('mainthread'). Dicha rutina es un método de Thread. Siempre que se efectúa una llamada a un método, la dirección de la instancia ligada a dicho método debe proporcionarse como primer parámetro (a pesar de que en llamadas de método a método este parámetro esté implícito). Puesto que es posible pasar un parámetro a la rutina de arranque de una 'thread', se pasa 'this' como -tal parámetro. De esta forma la rutina de arranque queda asociada a la misma instancia que la rutina de creación (ver la llamada a pthread_create en el código). La rutina de arranque llama entonces a la rutina virtual 'execute'. Como que la misma rutina es usada para todas las 'threads', se da un número de 'thread' correlativo como parámetro de 'execute'.

```
// Thread.h
// Thread class: thread launcher.
// Objects willing to start threads must inherit from this class
//-----------------------------------------------------------
#ifndef _Thread
#define _Thread
#include <pthread.h>
class Thread {

public:
  Thread (                    // Constructor
    int nb);                  // Number of threads

protected:
  void exit_thread (void);  // Exit from thread
  virtual void execute (    // Thread execute function
    int num_thread) = 0;  // Thread number. Threads started from
                          // a single object get here a correlative
                          // number (starting from 1).

private:
  int mainthread (          // Thread main routine
    Thread *thread);        // The instance from where the thread
                                          is started
  int nb_thread;            // Number of the last started thread
};
#endif
```

```
// Thread.cc
//    Thread class: thread launcher
//-----------------------------------------------------------
#include <stdlib.h>
#include <pthread.h>
#include «Thread.h»

// Constructor
//-----------------------------------------------------------
Thread::Thread (
  int nb)                    // number of threads
  : nb_thread (0)
{
  pthread_attr_t attr;       // thread attributes
  pthread_t tidp;            // thread identifier
                             // creation of threads
  pthread_attr_create (&attr);
  for (int i = 0; i < nb; i++) {
    if (pthread_create (&tidp, attr, mainthread, this) == -1) {
      perror («pthread_create»);
      exit (-1);
    }
  }
}

// Thread exit
//-----------------------------------------------------------
void Thread::exit_thread (void)
{
  pthread_exit (0);
}

// Main routine of the created thread
//-----------------------------------------------------------
Thread::mainthread (
  Thread *thread)            // 'this' is here
{
                             // especific routine call
  execute (++nb_thread);
}
```

3. Ejemplo

En el ejemplo que sigue se crean diversas 'threads', que se activan mutuamente y compiten para acceder a unos datos comunes. Se ha programado usando los objetos descritos en el apartado precedente.

Existen tres clases y se crea una instancia para cada una de ellas. Por razones de simplificación, las instancias se crean como variables globales.

La clase Entity tiene datos privados que pueden ser accedidos en un entorno multithread. Por esta razón, el acceso a estos datos privados se protege por medio de un 'mutex'. Este 'mutex' es a su vez un atributo privado.

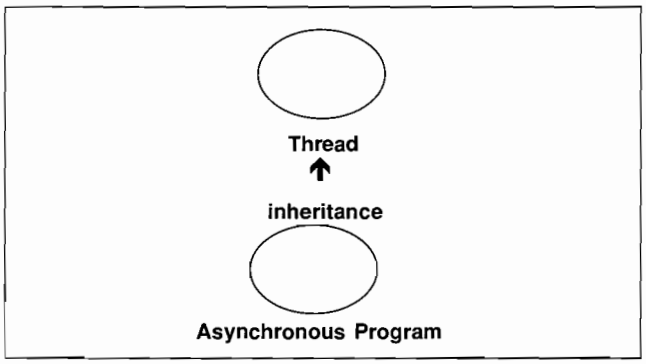


Figura 3

La clase Trace implementa una 'thread'. También define una variable de condición. Ambos elementos, la variable de condición y la rutina de ejecución de la 'thread' ('execute') son privados. La rutina de ejecución de la 'thread' se pone en espera de la variable de condición con una temporización. Cuando se activa, sea por una señal explícita sea por fin de temporización, lee el dato de Entity y lo guarda en un fichero. El método público 'activate' envía una señal a la variable de condición, activando en consecuencia la 'thread'.

La clase Interface también implementa una 'thread'. Esta 'thread' lee caracteres desde la consola y los guarda en Entity. Luego activa la 'thread' contenida en Trace con la llamada adecuada **(Figura 4)**. La rutina principal ejecuta la 'thread' principal del proceso. Se trata de una rutina vacía.

```
// TestThread.cc
// Thread encapsulation test program.
//-----------------------------------------------------------
#include <stdlib.h>
#include <pthread.h>
#include «Thread.h»
#include «Mutex.h»
#include «Condition_var.h»

// Entity class.
//  It has two private variables: char1 and char2.
//  The 'get' method gets them.
// The 'store' method puts char1 in char2, and a parameter i char1
//  The access to the variables is protected by a mutex.
//-----------------------------------------------------------

class Entity {

public:
  Entity (void)              // Constructor
    : char1 ('0'), char2 ('0') {};
  get (char *p_char1, char *p_char2) // get method
  {
    mutex.lock ();
    *p_char1 = char1;
    *p_char2 = char2;
    mutex.unlock ();
  }

  store (char newchar)       // store method
  {
    mutex.lock ();
    char2 = char1;
     char1 = newchar;
    mutex.unlock ();
  }

private:
     char char1, char2;      // Variables
    Mutex mutex;             // Mutex
};

Entity entity;               // Global instance declaration

// Trace class.
//  It defines a thread. This thread waits on a timed condition variable.
When
//    it is activated, either because of an explicit activation or after a
time
//    out, it gets char1 and char2, and writes them down in a file.
//    Character 'z' ends the thread.
```

```
//     The 'activate' method activates the thread.
//-------------------------------------------------------
class  Trace
  : public Thread {

public:
  Trace (void)            // Constructor
    : Thread (1) {};
                          // Activate, unless the activation flag
                          // is already set. This flag is set until
                          // the thread waits again.
  void activate (void)
  {
    cond.lock ();
    if (activation_fl == 0) {
      activation_fl = 1;
      cond.signal ();
    }
    cond.unlock ();
  };

private:
  void execute (int num)          // Thread routine
  {
    FILE *fd;
    char c1, c2;
    int tout;
    fd = fopen («data», «w»);
    for (;;) {
                                  // Wait for condition and
                                  // activation flag or time out
      cond.lock ();
      activation_fl = 0;
      while (activation_fl == 0) {
        if ((tout = cond.wait (2000)) == 0) {
          activation_fl = 1;
        }
      }
      cond.unlock ();
                          // Get and write data
      entity.get (&c1, &c2);
      fprintf (fd, "char1 = %c, char2 = %c, timeout condition = %d\n",
        c1, c2, tout);
      if (c1 == 'z') {
        fclose (fd);
        exit_thread ();
      }
    }
  };
  Condition_var cond;             // Condition variable
  int activation_fl;              // Activation flag
};

Trace trace;                      // Global instance declaration

// Interface class.
// It defines a thread. This reads characters from the console.
// For each character, it sets char1 and char2.
// Character 'z' ends the thread.
//-------------------------------------------------------

class  Interface

  : public Thread {
public:
  Interface (void)                // Constructor
    : Thread (1) {};

private:
  void execute (int num)          // Thread routine
  {
    char string [10];
                                  // Get character and store it
    for (;;) {
      printf («Enter character and CR:»);
      gets (string);
      entity.store (string [0]);
                                  // Activate trace
      trace.activate ();
      if (string [0] == 'z') {
        exit_thread ();
      }
    }
  }
};

Interface interface;              // Global instance declaration

// main routine.
//-------------------------------------------------------
main ()
{
  pthread_exit (0);
}
```

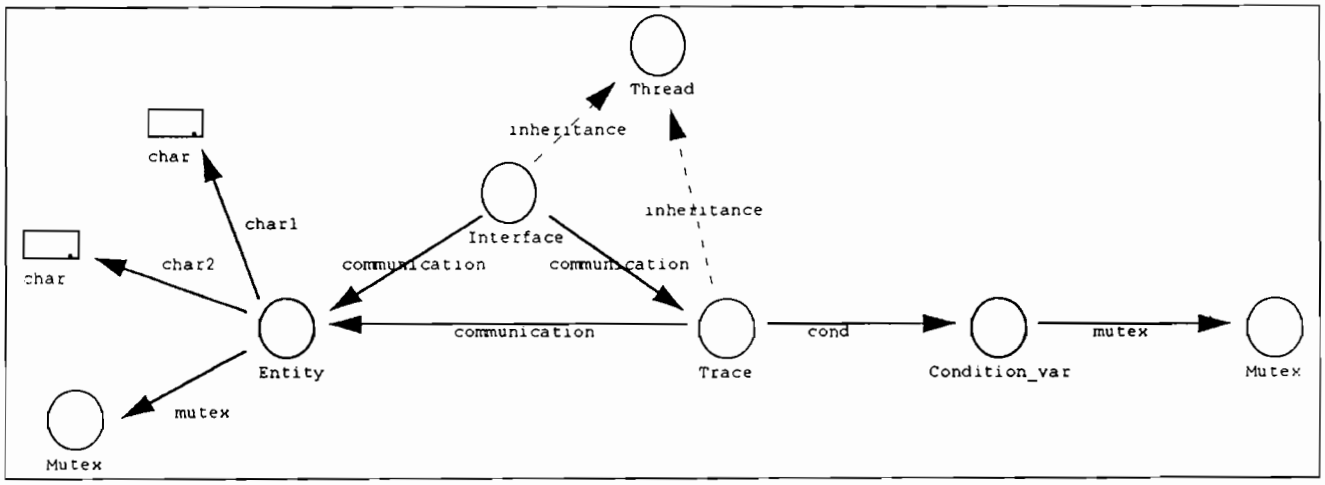


Figura 4

Enseñanza universitaria de la informática

J. Ángel Velázquez Iturbide
Dpto. Lenguajes y Sistemas Informáticos e Ingeniería de Software, Universidad Politécnica de Madrid

Un problema combinatorio y su resolución con cinco técnicas algorítmicas

Resumen: *la resolución de un mismo problema mediante varias técnicas de diseño es un método útil para enseñar las principales características, ventajas y desventajas de cada técnica en una asignatura de algoritmos. Ilustramos este método resolviendo un problema combinatorio, el problema de la mochila 0/1, mediante cinco técnicas algorítmicas: recursividad; retroceso; memorización; tabulación; ramificación y acotación. Incluimos una comparación de su eficiencia, basada tanto en análisis de complejidad como en medidas de tiempo.*

1. Introducción

Aunque el conjunto de materias que se consideran básicas en la formación de un informático ha ido creciendo con los años, el diseño y análisis de algoritmos sigue siendo una parte fundamental de cualquier plan de estudios (véase p.ej. [12]). En nuestra facultad, la asignatura de segundo curso llamada *Programación I* es la encargada de este cometido [6]. Una parte importante de la misma se dedica a estudiar técnicas de diseño de algoritmos.

Un instrumento útil para enseñar técnicas de diseño de algoritmos es la resolución de un mismo problema mediante varias de estas técnicas. Al usarse el mismo problema, no se desvía la atención con los detalles particulares de un nuevo problema, y por tanto se facilita la exposición y comprensión de cada técnica.

El propósito de este artículo es ejemplificar este método didáctico con un problema resoluble por una amplia variedad de técnicas. Una clase de problemas adecuada para este propósito es la de los problemas combinatorios de optimización, de los cuales hemos elegido el *problema de la mochila 0/1*. Justificamos su elección con que: (a) es un problema ampliamente conocido y tratado en numerosos textos (p.ej. [7, 9]), que así pueden consultarse, y (b) tiene un grado de dificultad medio, que permite mostrar las características de cada técnica sin excesiva complejidad. Un comentario pertinente es que, aunque hemos intentado citar en el artículo los aspectos más importantes de las técnicas usadas, no tenemos espacio para explicarlos más detalladamente, por lo que remitimos al lector interesado a la bibliografía adjunta.

El enunciado del problema es el siguiente. Se dispone de una mochila que soporta un peso máximo c, que denominamos su capacidad. También se tienen n objetos tales que el objeto i-ésimo tiene un peso ps_i y proporciona un beneficio bs_i al meterlo entero en la mochila (seguimos un criterio corriente en programación funcional por el que una secuencia se expresa en plural; en concreto, una secuencia de pesos se simboliza ps en vez de p, por lo que el elemento ps_i es un peso particular). La capacidad de la mochila y los pesos y beneficios de los objetos son números naturales. El problema consiste en rellenar la mochila de forma que se obtenga un beneficio máximo, que no se sobrepase su capacidad y que los objetos sólo se metan enteros. Por ejemplo, sea un caso particular con cuatro objetos en que c=15, $(ps_1,ps_2,ps_3,ps_4) = (3,6,9,5)$ y $(bs_1,bs_2,bs_3,bs_4) = (7,2,8,4)$. El lector puede comprobar que hay muchas formas posibles de rellenar la mochila. La solución óptima consiste en tomar el primer y tercer objeto, con beneficio 15. En lo sucesivo usamos este ejemplo.

Conviene expresar el problema en el lenguaje de programación que usemos, en nuestro caso Modula-2. Sean las siguientes declaraciones:

```
CONST
    N = ...;
    C = ...;
TYPE
    PESO        = [0..C];
    BENEFICIO   = INTEGER;
    ETAPA       = [1..N+1];
    PESOS       = ARRAY ETAPA OF PESO;
    BENEFICIOS  = ARRAY ETAPA OF BENEFICIO;
```

La cabecera de un procedimiento que resuelve el problema es:

```
PROCEDURE   Mochila01 (ps: PESOS;
                       bs: BENEFICIOS;
                       c : PESO;
                       VAR b: BENEFICIO);
```

Obsérvese que el procedimiento sólo calcula el beneficio óptimo, pero para que fuera práctico debería ampliarse con otro parámetro de salida que represente la decisión asociada. No lo incluimos para no complicar los algoritmos presentados, pero el lector interesado puede consultar [13].

En el resto del artículo se tratan los siguientes puntos. El apartado segundo muestra dos algoritmos sencillos pero ineficientes: uno funcional y otro de retroceso. El tercer apartado muestra estos algoritmos mejorados: el recursivo da lugar a un algoritmo memorizador y a otro tabulado; el de retroceso, a un algoritmo de ramificación y acotación. En el apartado cuarto se incluyen algunas medidas tomadas experimentalmente de sus tiempos de ejecución, que complementan los análisis de complejidad hechos en los apartados segundo y tercero. Se termina dando una pequeña bibliografía recomendada.

2. Algoritmos sencillos

Una característica importante de los problemas combinatorios es que sus soluciones son compuestas. Existe una forma sistemática de resolver problemas combinatorios de

optimización: se generan todas las soluciones posibles y se busca la óptima. Normalmente las soluciones potenciales se construyen paso a paso y de forma tal que se mejora la eficiencia de su construcción, p.ej. se mezcla la construcción con las comprobaciones de validez u optimidad.

Hay dos elementos importantes para el diseño de estos algoritmos:

a) ***Construcción de la solución compuesta por etapas.*** Primero se formula una secuencia de etapas, tales que en cada una se decide el valor de un componente de la solución. Segundo se identifican los valores que son candidatos válidos en cada etapa, es decir, como valores de cada componente. Ambos aspectos quedan claramente expuestos en una representación gráfica como los árboles de búsqueda.

Volviendo al problema de la mochila 0/1, una solución puede representarse como una secuencia xs de n decisiones sobre los objetos. La decisión xs_i de introducir o no el objeto i-ésimo en la mochila puede representarse con el valor respectivo 1 ó 0 (de ahí el nombre). El objetivo del problema es encontrar una solución que maximice el beneficio total satisfaciendo la restricción de no sobrepasar la capacidad de la mochila:

$$maximizar \sum_{j=1}^{n} bs_j\ xs_j\ \ tal\ que \sum_{j=1}^{n} ps_j\ xs_j \le c \quad xs_j \in \{0,1\},\ 1 \le j \le n$$

La solución óptima para nuestro ejemplo consiste en introducir los objetos primero y tercero, que se representa así $(xs_1,xs_2,xs_3,xs_4)=(1,0,1,0)$.

Podemos construir sistemáticamente las soluciones válidas del problema con un árbol de decisión booleana. Definimos tantos niveles de nodos como objetos hay en el enunciado del problema. Un nodo perteneciente al nivel i, $1 \le i \le n$, representa la situación alcanzada tras tomar ciertas decisiones sobre la inclusión en la mochila de los objetos anteriores $1..i$-1. En este estado sólo hay dos decisiones posibles para el objeto i-ésimo: bien se deja fuera de la mochila, bien se mete dentro. Por último, se añade un nivel n+1 de nodos para completar el árbol, pero en ellos no se realiza ninguna tarea útil. Una solución válida es la secuencia xs de valores que se obtienen recorriendo un camino del árbol desde el nodo de raíz hasta un nodo de hoja. La solución óptima es la solución válida que proporcione un beneficio mayor.

Volvamos a nuestro ejemplo. Si cada nodo se etiqueta con la capacidad aún libre de la mochila, el árbol de decisión correspondiente se muestra en la **figura 1**, donde la solución

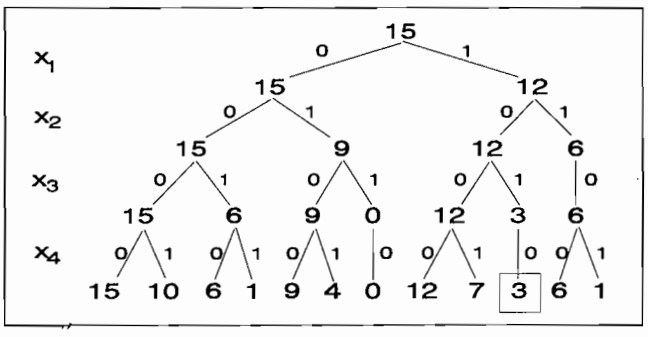


Figura 1: Árbol binario de decisión

óptima se obtiene recorriendo el camino que va desde el nodo de raíz hasta el nodo de hoja enmarcado. De algunos nodos sólo surge una decisión 0: esto sucede cuando el objeto en consideración no cabe en la mochila.

b) ***Generalización del problema original.*** En problemas de optimización se generaliza el problema original para que se puedan representar los subproblemas de etapas intermedias. De esta forma puede expresarse que una solución se construye a partir de sus subsoluciones.

Recordemos que el problema de la mochila 0/1 consiste en:

$$maximizar \sum_{j=1}^{n} bs_j\ xs_j\ \ tal\ que \sum_{j=1}^{n} ps_j\ xs_j \le c \quad xs_j \in \{0,1\},\ 1 \le j \le n$$

Veamos una generalización sencilla basada en el árbol de decisión anterior. Supongamos que se han tomado decisiones sobre los objetos $1..i$-1. El primer objeto por considerar es uno cualquiera i, $1 \le i \le n$, y la capacidad aún libre de la mochila es p, $0 \le p \le c$. Decimos que $m(i,p)$ es la representación del problema restante:

$$maximizar \sum_{j=1}^{n} bs_j\ xs_j\ \ tal\ que \sum_{j=1}^{n} ps_j\ xs_j \le p \quad xs_j \in \{0,1\},\ i \le j \le n$$

El problema completo de la mochila 0/1 coincide con el subproblema $m(1,c)$. Los subproblemas deben cumplir ciertas relaciones entre sí para aplicar algunas técnicas de diseño de algoritmos. Así, el principio de optimidad es necesario para aplicar programación dinámica [7] y cierta implicación entre condiciones de validez se requiere para la técnica de retroceso [9]. El problema de la mochila 0/1 cumple ambas relaciones.

2.1. Algoritmo funcional

A partir de las decisiones de diseño anteriores, es muy sencillo obtener un algoritmo funcional. El caso básico se da en la etapa n+1, es decir, cuando no queda ningún objeto que considerar:

$$m(n+1,p)=0$$

para una capacidad libre p cualquiera.

Veamos el caso recursivo cuando se considera un objeto genérico i y la mochila tiene libre cierta capacidad p. Obviamente hay dos posibilidades, dejar el objeto i fuera o meterlo en la mochila, es decir, $xs_i=0$ ó 1. En el primer caso, dicho objeto no añade peso a la mochila ni tampoco proporciona ningún beneficio. La solución es el beneficio máximo obtenible con los demás objetos, es decir, $m(i,p) = m(i+1,p)$. En caso de que el objeto i se meta en la mochila, hay que considerar el aumento de beneficio y la disminución de capacidad libre que produce el objeto; es decir, $m(i,p) = bs_i + m(i+1,p\text{-}ps_i)$. Entre ambas soluciones, debe tomarse la que proporcione mayor beneficio:

$$m(i,p)=\max(m(i+1,p),bs_i+m(i+1,p\text{-}ps_i))$$

Sin embargo, el caso recursivo se complica algo al tener en cuenta la restricción de que no se sobrepase la capacidad de la mochila. Si un objeto tiene un peso mayor que la capacidad libre de la mochila, no puede meterse, por lo que la única elección posible es descartarlo. Luego:

$$m(i,p) = \begin{cases} m(i+1,p) & \text{si } p<ps_i \\ \max(m(i+1,p), bs_i + m(i+1,p-ps_i)) & \text{en caso contrario} \end{cases}$$

El algoritmo aparece en la **figura 2** escrito en Modula-2, conservando el estilo funcional y la notación $m(i,p)$ usada para los subproblemas.

El algoritmo es sencillo, pero tiene el inconveniente de ser terriblemente ineficiente en tiempo. El caso mejor se da cuando todos los objetos tienen un peso mayor que la capacidad de la mochila. En este caso, siempre se toma la primera rama recursiva, con lo que la complejidad del algoritmo es lineal $O(n)$. El caso peor se da cuando la suma de los pesos de todos los objetos es menor que la capacidad de la mochila. Entonces se toma siempre la segunda rama recursiva, que contiene dos llamadas recursivas; dado que hay n objetos, la complejidad resultante es exponencial $O(2^n)$.

Para calcular la eficiencia en espacio, obsérvese que el algoritmo necesita un espacio constante durante cada llamada recursiva. Por tanto, la máxima cantidad de memoria ocupada es proporcional a la profundidad máxima del árbol de recursión, que es $O(n)$.

2.2. Algoritmo de retroceso

El árbol de decisión diseñado al comienzo del apartado ha sido la base para la estructura recursiva de la función diseñada, pero también puede ser la base para un algoritmo de retroceso. El algoritmo de retroceso se obtiene a partir del funcional con sólo hacer algunas transformaciones. Primero, no es una función sino un procedimiento con un parámetro que va propagando la mejor solución encontrada. Segundo, la estructura recursiva se modifica ligeramente para evitar las inútiles llamadas recursivas del último nivel. Tercero, en vez de aparecer explícitamente las llamadas recursivas correspondientes a todos los candidatos, se enmascaran en un bucle FOR que las produce en sucesivas iteraciones.

En nuestro algoritmo, el parámetro propagado es el máximo beneficio encontrado. Se incluye un parámetro adicional que acumula el beneficio de la solución en formación; su misión es que el beneficio de cada solución válida se calcule eficientemente. Los candidatos de cada nivel son los códigos usados para representar las decisiones sobre el objeto i, que son 0 y 1; por tanto, el bucle FOR itera sobre estos dos valores.

```
PROCEDURE Mochila01 (ps : PESOS;
                     bs : BENEFICIOS;
                     c  : PESO;
                     VAR b : BENEFICIO);

   PROCEDURE MochilaFuncional (i : ETAPA;
                               p : PESO) : BENEFICIO;
      BEGIN
         IF i = N+1 THEN
           RETURN 0
         ELSIF p < ps[i] THEN
           RETURN MochilaFuncional (i+1,p)
         ELSE
           RETURN Max (MochilaFuncional(i+1,p),
                       bs[i] + MochilaFuncional(i+1,p-ps[i]))
         END
      END MochilaFuncional;

   BEGIN
     b := MochilaFuncional (1, c)
   END Mochila01;
```

Figura 2: Algoritmo funcional

Modificando el algoritmo anterior con estas ideas resulta el algoritmo de la **figura 3**.

Las modificaciones realizadas son detalles de codificación que no afectan a las partes clave del algoritmo, por lo que éste tiene la misma complejidad en espacio y en tiempo que el funcional.

3. Algoritmos eficientes

Los dos algoritmos del apartado anterior tienen una complejidad exponencial en el caso peor, es decir, son muy ineficientes. Que esta ineficiencia es evitable se comprueba determinando el número de subproblemas existentes. Si un problema genérico de la mochila 0/1 se representa mediante la notación $m(i,p)$ definida en el apartado anterior, hay $n+1$ posibles valores de i ($1 \pounds i \pounds n+1$) y $c+1$ posibles valores de p ($0 \pounds p \pounds c$); por tanto, el número total de subproblemas distintos es $(n+1)(c+1)$. Dado que hay $O(cn)$ subproblemas distintos y que $O(2^n) >> O(cn)$, la única explicación posible para que los algoritmos anteriores calculen $O(2^n)$ subproblemas es que algunos subproblemas se estén resolviendo repetidas veces.

Hay varias formas de mejorar la eficiencia de un algoritmo redundante; aquí mostramos dos. Ambos algoritmos son básicamente iguales, pero sus estilos de programación no, por lo que cada uno se mejora más fácilmente con una técnica diferente:

a) Cada subproblema se calcula una sola vez y se almacena para su eventual consulta en una tabla. Las técnicas de tabulación son más fácilmente aplicables cuanto más sencilla es la definición de la función recursiva original. Aun así, hay varias técnicas de tabulación, de las que vamos a ver las llamadas memorización y sobretabulación (o tabulación). Los algoritmos resultantes de esta última clase de tabulación son los conocidos con el nombre de programación dinámica.

b) Se abandona la búsqueda por aquellos subproblemas que se sabe que no conducen a una solución óptima. Esto se hace más fácilmente sobre algoritmos de retroceso mediante técnicas de poda, produciéndose los algoritmos llamados de ramificación y acotación.

```
PROCEDURE Mochila01 (ps : PESOS;
                     bs : BENEFICIOS;
                     c  : PESO;
                     VAR b: BENEFICIO);

   PROCEDURE BuscarMochila (i  : ETAPA;
                            p  : PESO;
                            ba : BENEFICIO; (* b. actual *)
                            VAR b: BENEFICIO);
      TYPE
         DECISION = [0..1];
      VAR
         x  : DECISION;
         np : PESO;      (* nuevo peso *)
         nb : BENEFICIO; (* nuevo beneficio *)
      BEGIN
         FOR x := 0 TO 1 DO
            IF p >= x*ps[i] THEN
               np := p - x*ps[i];
               nb := ba + INTEGER(x)*bs[i];
               IF i = N THEN
                  b := Max (nb, b)
               ELSE
                  BuscarMochila (i+1, np, nb, b)
               END;
            END;
         END;
      END BuscarMochila;
   BEGIN
      b := 0;
      BuscarMochila (1, c, 0, b)
   END Mochila01;
```

Figura 3: Algotirmo de retroceso

3.1. Algoritmo memorizador

La memorización es una forma de tabulación en la que el algoritmo imperativo conserva la estructura recursiva del algoritmo funcional. La tabla contiene tantas entradas como subproblemas hay. Inicialmente, la tabla no contiene el valor de ningún subproblema. Cuando se resuelve por primera vez un subproblema, su valor se almacena (se memoriza) en la tabla; cuando en lo sucesivo se necesite dicho valor, no se recalcula sino que se extrae de la tabla. La tabla que usemos para memorizar un problema debe ser capaz de realizar dos tareas:

a) ***Determinar si se ha almacenado el valor de cierto subproblema, y en caso afirmativo conocer su valor.*** En el problema de la mochila 0/1, el beneficio de un subproblema resuelto es un número natural. Por tanto, aquellos subproblemas aún no resueltos pueden identificarse con un beneficio ficticio negativo, p.ej. el número -1.

b) ***Almacenar los valores de los subproblemas necesarios para calcular un problema dado.*** La tabla debe tener tantos elementos como subproblemas existan, para poder almacenar su valor. Nuestro problema se ha denotado $m(i,p)$, $1 \pounds i \pounds n+1$, $0 \pounds p \pounds c$, por lo que necesitamos una tabla con $(n+1)(c+1)$ elementos. La tabla puede repre-sentarse de forma sencilla mediante una matriz bidimensional de $(n+1)(c+1)$ elementos y con el mismo número y rango de índices que la notación usada para formalizar el problema.

El algoritmo memorizador aparece en la **figura 4**. En la **figura 5** se muestra el estado de la tabla tras la ejecución del algoritmo sobre nuestro ejemplo. Las celdas vacías contienen el valor indefinido -1. Las flechas indican la propagación de valores al retornar las llamadas recursivas. El beneficio máximo es el contenido final de la celda $(1,c)$, que es 15. Obsérvese que se pregunta dos veces por el subproblema $m(4,6)$. La primera vez es llamado por el subproblema $m(3,15)$; su valor 4 se calcula mediante llamadas recursivas y se almacena. Cuando la segunda vez es llamado por $m(3,6)$, sólo se extrae el valor almacenado.

Puede comprobarse que su complejidad en tiempo en el caso peor es $O(nc)$, por lo que es un resultado eficiente. Sin embargo, su complejidad en espacio aumenta debido a la tabla, siendo $O(nc)$.

3.2. Algoritmo tabulado

La técnica de memorización permite resolver problemas eficientemente en tiempo. Ahora bien, el algoritmo reserva memoria para todos los subproblemas, aunque sólo se determine el valor de algunos; también necesita memoria adicional para la pila de control usada por el proceso recursivo. El resultado es un algoritmo ineficiente en espacio.

Los algoritmos redundantes pueden tabularse más eficientemente en memoria si se analiza la forma de su redundancia. Se dibuja el árbol de recursión del algoritmo correspondiente a una aplicación concreta. Uniendo todos los nodos que representan una misma llamada recursiva, se obtiene un grafo que muestra la dependencia entre distintas llamadas recursivas (es decir, subproblemas). El grafo de dependencia puede ser más o menos uniforme, lo cual conduce a distintos grados de eficiencia en la tabulación. A partir del grafo se determina un orden lineal de cómputo de los subproblemas necesarios para resolver el problema completo. Este nuevo orden de cómputo se implementa mediante un algoritmo iterativo. También se determina el tamaño mínimo de la tabla, que será igual al número mínimo de subproblemas necesarios para seguir calculando los subproblemas aún pendientes.

Recordemos las ecuaciones de nuestro algoritmo funcional:

$$m(n+1,p) = 0$$

$$m(i,p) = \begin{cases} m(i+1,p) & \text{si } p<ps_i \\ \max(m(i+1,p), bs_i + m(i+1,p-ps_i)) & \text{en caso contrario} \end{cases}$$

A priori no se sabe qué subproblemas se necesitan para resolver el problema $m(1,c)$, ya que depende del contenido del vector ps de pesos. Sin embargo, es fácil observar que cualquier problema $m(i,p)$ sólo depende de subproblemas $m(i+1,p')$, donde $\leq p' \leq p$.

Una solución iterativa adecuada para las ecuaciones de recurrencia anteriores consiste en calcular los subproblemas por filas. Se parte del caso básico, $i=n+1$, y se resuelven sucesivamente todos los subproblemas con $i=n,...,1$. Las celdas rellenas en la primera iteración contendrán el valor 0,

```
PROCEDURE Mochila01 (ps : PESOS;
                     bs : BENEFICIOS;
                     c  : PESO;
                     VAR b : BENEFICIO);
  VAR
     tabla : ARRAY ETAPA, PESO OF BENEFICIO;

  PROCEDURE MochilaConMemoria (i : ETAPA;
                               p : PESO): BENEFICIO;
    BEGIN
      IF tabla[i,p] = -1 THEN
        IF i = N+1 THEN
          tabla[i,p] := 0
        ELSIF p < ps[i] THEN
          tabla[i,p] := MochilaConMemoria (i+1,p)
        ELSE
          tabla[i,p] :=
                Max (MochilaConMemoria (i+1,p),
                     bs[i] + MochilaConMemoria(i+1,p-ps[i]))
        END
      END;
      RETURN tabla[i,p]
    END MochilaConMemoria;

  VAR
     i : ETAPA;
     p : PESO;
  BEGIN
     FOR i := 1 TO N+1 DO
        FOR p := 0 TO c DO
           tabla[i,p] := -1
        END
     END;
     b := MochilaConMemoria (1, c)
  END Mochila01;
```

Figura 4: Algoritmo memorizador

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1																15
2													8			12
3							4			8			8			12
4	0			0			4			4			4			4
5	0	0		0	0		0	0		0	0		0			0

Fig. 5 Contenido de la tabla tras ejecutar el algoritmo memorizador

```
PROCEDURE Mochila01 (ps : PESOS;
                     bs : BENEFICIOS;
                     c  : PESO;
                     VAR b : BENEFICIO);
   VAR
      tabla : ARRAY PESO OF BENEFICIO;
      i : ETAPA;
      p : PESO;
   BEGIN
      FOR p := 0 TO c DO
         tabla[p] := 0
      END;
      FOR i := N TO 1 BY -1 DO
        FOR p := c TO ps[i] BY -1 DO
           tabla[p] := Max (tabla[p], bs[i] + tabla[p-ps[i]])
        END
      END;
      b := tabla[c]
   END Mochila01;
```

Figura 6: Algoritmo tabulado

correspondiente a $m(n+1,p)$, para $0 \leq p \leq c$. Al comienzo de una iteración una fila contiene el valor de todos los subproblemas $m(i+1,p)$, y a su término la fila siguiente contendrá todos los subproblemas $m(i,p)$, $0 \leq p \leq c$. La solución del problema se encuentra al final del algoritmo en la celda $(1,c)$.

Con el proceso explicado puede llenarse una matriz de dimensión $(n+1)(c+1)$, pero puede reducirse el tamaño de la tabla examinando el tiempo de vida de la solución a cada subproblema. Puesto que cada problema $m(i+1,p)$ sólo se usa para calcular subproblemas $m(i,p')$, los subproblemas de una fila sólo necesitan guardarse para calcular la fila siguiente. Si cada iteración actualiza las celdas en orden decreciente de pesos, basta con un solo vector: el lector puede comprobar que una vez calculado un valor $m(i,p)$, el valor $m(i+1,p)$ ya no se necesita más, por lo que aquél puede almacenarse donde se encontraba éste.

La implementación del algoritmo es sencilla. Obsérvese que el algoritmo queda muy conciso haciendo variar en cada iteración el índice p desde c hasta ps_i, en vez de llegar hasta 0. El algoritmo tabulado resultante se encuentra en la **figura 6**.

En la **figura 7** se muestra para nuestro ejemplo el estado del vector inicialmente y tras cada iteración. Recuérdese que las iteraciones progresan con i =4,3,2,1, es decir, de abajo hacia arriba en la figura. La línea gruesa de cada fila muestra el extremo izquierdo del subvector tratado durante la iteración correspondiente.

El algoritmo tiene una complejidad en tiempo $O(nc)$ similar a la del algoritmo memorizador, y es difícil decir cuál es más rápido. El algoritmo tabulado calcula todos los subproblemas existentes, con independencia de que se necesiten para $m(1,c)$, por lo que calcula más subproblemas que el algoritmo memorizador. Sin embargo, el algoritmo memorizador tiene el coste de la recursividad y la iniciación de la tabla. En cambio, es evidente que la complejidad en espacio del algoritmo tabulado ha disminuido a lineal $O(c)$.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	7	7	7	7	7	11	11	11	11	15	15	15	15
2	0	0	0	0	0	4	4	4	4	8	8	8	8	8	12	12
3	0	0	0	0	0	4	4	4	4	8	8	8	8	8	12	12
4	0	0	0	0	0	4	4	4	4	4	4	4	4	4	4	4
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 7 Contenido de la tabla tras ejecutar el algoritmo tabulado

3.3. Algoritmo de ramificación y acotación

El algoritmo de retroceso planteado en el apartado segundo podría mejorarse con las dos técnicas de tabulación recién expuestas. Sin embargo, es más usual que los algoritmos de retroceso se mejoren mediante una reducción del tamaño del árbol de búsqueda. Hay múltiples maneras de reducir este tamaño, siendo la conocida como ramificación y acotación específica de problemas de optimización.

Un algoritmo de ramificación y acotación se basa en asociar una medida auxiliar a cada nodo del árbol de búsqueda. Esta medida es una estimación del valor óptimo obtenible a partir de dicho nodo. En un problema de maximización como el que nos ocupa, la medida del nodo inicial es un valor alto. Además, debe cumplirse la propiedad de que un nodo descendiente tenga un valor asociado menor o igual al de su nodo antecesor.

Los algoritmos de ramificación y acotación usan la medida para abandonar cualquier rama del árbol de búsqueda que no pueda conducir a una solución óptima. Estos algoritmos propagan el valor v de la mejor solución encontrada. Si se encuentra un nodo cuyo valor es menor o igual que v, la propiedad anterior asegura que cualquier nodo descendiente también tendrá un valor menor o igual que v. Como nuestro objetivo es maximizar, ningún descendiente proporcionará una solución mejor al problema y por tanto podemos abandonar la búsqueda por dicho nodo y sus descendientes. El valor de la medida estimada se llama *cota* y abandonar una rama del árbol de búsqueda se llama *poda* de la rama.

Con frecuencia pueden definirse diversas medidas para un mismo problema. Hay dos criterios importantes para elegir una medida concreta. Uno, la bondad de la aproximación que la

```
PROCEDURE Mochila01 (ps : PESOS;
                     bs : BENEFICIOS;
                     c  : PESO;
                     VAR b: BENEFICIO);

   PROCEDURE BuscarMochila (i  : ETAPA;
                            p  : PESO;
                            ba : BENEFICIO; (* ben. actual *)
                            VAR b: BENEFICIO);
      VAR
         x    : [0..1];
         nb   : BENEFICIO; (* nuevo beneficio *)
         np   : PESO;      (* nuevo peso *)
         cota : REAL;
      BEGIN
         FOR x := 0 TO 1 DO
            IF p >= x*ps[i] THEN
               np := p - x*ps[i];
               nb := ba + INTEGER(x)*bs[i];
               IF i = N THEN
                  cota := REAL(nb)
               ELSE
                  cota := REAL(nb) +
                          REAL(np)*REAL(bs[i+1])/REAL(ps[i+1])
               END;
               IF INTEGER(cota) > b THEN
                  IF i = N THEN
                     b := nb
                  ELSE
                     BuscarMochila (i+1, np, nb, b)
                  END;
               END;
            END;
         END;
      END BuscarMochila;

   BEGIN
      Ordenar (ps, bs);
      b := 0;
      BuscarMochila (1, c, 0, b)
   END Mochila01;
```

Figura 8: Algoritmo de ramificación y acotación

medida proporciona al valor óptimo obtenible desde el nodo. Segundo, el tiempo necesario para calcular la medida. Conviene elegir medidas que sean buenas aproximaciones de los valores óptimos obtenibles y que se calculen en poco tiempo.

Veamos su aplicación al problema de la mochila 0/1. Supóngase que se han tomado decisiones sobre los objetos 1..*i* con un beneficio *b*. El máximo beneficio que puede obtenerse desde este estado es dicho beneficio *b* más el máximo beneficio obtenible con los demás objetos *i*+1..*n*. Este segundo componente del máximo beneficio es desconocido, por lo que debe estimarse.

El problema de la mochila, que es una versión del problema en la que los objetos pueden meterse partidos, proporciona diversas estimaciones. Supóngase que los objetos están en orden decreciente de tasa beneficio/peso. Una estimación, mala pero fácil y eficiente de calcular, consiste en llenar completamente la mochila con el primer objeto aún no considerado. Otra estimación mejor la proporciona el algoritmo voraz [7, 9], que mete todo lo posible del primer objeto (porque da el mayor beneficio por unidad de peso), luego del segundo y así sucesivamente hasta que la mochila se llena. El algoritmo resultante de adoptar la primera estimación ha aparecido en la **figura 8**.

La complejidad de un algoritmo de ramificación y acotación sigue siendo exponencial en el caso peor, pero si la medida usada es buena, en la práctica se reduce mucho el tamaño del árbol de búsqueda generado. Conviene medir tiempos de ejecución para comprobar la bondad de la medida elegida.

4. Comparación de rendimientos

En los dos apartados anteriores hemos analizado la complejidad en tiempo y en espacio de los cinco algoritmos. Centrándonos en el tiempo, los peores algoritmos son el funcional y el de retroceso y los mejores, los dos tabulados. El algoritmo de ramificación y acotación será bueno si poda lo suficiente el árbol de búsqueda. Conviene medir tiempos para conocer experimentalmente la magnitud de los factores constantes que intervienen en estos algoritmos, y en el caso del de ramificación y acotación, para conocer la bondad de la medida acotadora.

Hemos programado los algoritmos expuestos con el entorno de Modula-2 de TopSpeed versión 3.02, y los hemos ejecutado en un ordenador personal 486 a 50MHz. Se han incluido dos algoritmos de ramificación y acotación: el primero es el antes expuesto [9], mientras que el segundo usa el algoritmo voraz de la mochila como medida acotadora e incluye algunas pequeñas mejoras [7].

Conviene advertir que los tiempos obtenidos son ilustrativos de la eficiencia en tiempo de los algoritmos, pero dependen bastante de los valores de los vectores de entrada. Si se quisieran tiempos más exactos, habría que planificar los experimentos más cuidadosamente. Como detalle importante de las condiciones del experimento, los contenidos de los vectores se han producido aleatoriamente con un generador de Lehmer [13]; los pesos se han limitado al margen 1..10. La ordenación inicial de los objetos en los algoritmos de ramificación y acotación se ha realizado con el algoritmo rápido de Hoare.

Puesto que hay dos tamaños que influyen en la eficiencia de los algoritmos, *n* y *c*, hemos realizado dos experimentos. En el primero, hemos mantenido la capacidad constante y se ha variado el número de elementos. Los resultados obtenidos en centésimas de segundo aparecen en la **figura 9**.

Pueden deducirse las siguientes afirmaciones:

a) Los tiempos medidos son proporcionales a los órdenes de complejidad calculados analíticamente, por lo que el número *n* de objetos demuestra tener una gran influencia en el tiempo de ejecución de algunos algoritmos.

b) Los factores constantes son importantes en algunos casos. Se ve claramente con los dos primeros algoritmos: aunque son casi iguales, el de retroceso tarda aproximadamente el doble que el funcional, debido a la sobrecarga introducida por diversas instrucciones (control del bucle, etc). Algo similar sucede con el algoritmo memorizador respecto al tabulado.

c) Los algoritmos exponenciales son los mejores para valores muy pequeños, pero para apenas *n*=10 ya son muy lentos.

d) Los dos algoritmos de ramificación y acotación tienen comportamientos muy diferentes, al estar basados en acotaciones distintas. El segundo es claramente mejor, manteniéndose en el orden de los tiempos de los dos algoritmos tabulados. Probablemente aún pueda mejorarse la medida acotadora.

Veamos en la **figura 10** los resultados de variar la capacidad de la mochila, para *n*=10. Algunas conclusiones son:

a) El tamaño del parámetro *c* influye mucho menos que *n* en los tiempos, como ya se había comprobado en los análisis de complejidad.

b) Existe un tamaño de *c* a partir del cual el tiempo de los algoritmos exponenciales se mantiene constante. Esto se debe a que un aumento de la capacidad de la mochila empeora

C=100 N	5	10	15	20	25	30
Funcional	0,0162	0,600	20,160	644,80	20.630	719.960
Retroceso	0,0272	0,930	29,930	955,10	30.560	970.310
Memorizador	0,0710	0,247	0,627	0,77	1,1	1,6
Tabulado	0,1160	0,236	0,357	0,44	0,5	0,5
Rami.y aco.1	0,0650	1,150	13,730	215,80	747	1.747
Rami.y aco.2	0,0552	0,440	0,990	3,30	5	17

Fig. 9: Tabla de tiempos I

N=10 C	10	20	40	60	100	500
Funcional	0,0615	0,336	0,604	0,616	0,616	0,61
Retroceso	0,1252	0,577	0,910	0,934	0,922	0,88
Memorizador	0,0566	0,132	0,186	0,230	0,240	0,55
Tabulado	0,0148	0,043	0,088	0,142	0,242	1,15
Rami. y aco.1	0,1489	0,297	0,450	0,724	0,724	1,98
Rami. y aco.2	0,1642	0,351	0,362	0,396	0,352	0,39

Fig. 10. Tabla de tiempos II

los tiempos hasta que la capacidad de la mochila es mayor que la suma de todos los pesos, momento en el que tenemos el caso peor y ya no importa que aumente más la capacidad.

c) Los algoritmos tabulados aumentan linealmente con *c*. Puede observarse que hay un valor de *c* a partir del cual los algoritmos tabulados tardan más que los exponenciales. Este valor es importante para aplicaciones prácticas del algoritmo.

d) El segundo algoritmo de ramificación y acotación produce tiempos buenos, lo que junto con los resultados del experimento anterior, lo convierte en el mejor algoritmo en general.

Bibliografía recomendada

Existe una amplia bibliografía sobre los temas tratados en el artículo, por lo que sólo daré algunas referencias (mis favoritas). En [7, 9] se exponen muy claramente diversas técnicas de resolución de problemas combinatorios, incluyendo la mayoría de las aquí tratadas. Otras buenas referencias de programación dinámica son [5, 11, 13]. En [14] se muestran varios ejemplos de retroceso, con énfasis en su esquema asociado y en representaciones eficientes. La optimización de los algoritmos de retroceso viene muy bien tratada en [3], comentando diversas formas de reducir el árbol de búsqueda. Sobre tabulación, el artículo de Bird [2] es fundamental; la memorización puede consultarse en [4, 5].

El lector interesado en el problema de la mochila 0/1 puede consultar también [7,9,13]. Hay numerosos libros que tratan el análisis de complejidad, de los que sólo citaremos [1,4,10]. Sin embargo, pocos tratan las medidas de tiempos, destacando [8,15].

Aunque el problema se ha planteado de forma que los beneficios son números naturales, no habría ningún problema en que fueran reales positivos. Sin embargo, si los pesos son reales, los algoritmos no pueden tabularse eficientemente. En este caso tenemos un problema inherentemente ineficiente, de los llamados NP-completos. En [7, 10] se trata cómo paliar su ineficiencia en la medida de lo posible.

Agradecimientos

Quiero dar las gracias a Ángel Sánchez Calle y especialmente a Cristóbal Pareja Flores por sus comentarios.

Referencias bibliográficas

[1] A. V. Aho, J. E. Hopcroft y J. D. Ullman; *"Data Structures and Algorithms"*, Addison-Wesley, 1983. Traducción al español de A. Vargas Villazón y J. Lozano Moreno: *Estructuras de datos y algoritmos*, Addison-Wesley Iberoamericana, 1988.

[2] R. Bird; *"Tabulation techniques for recursive programs"*, ACM Computing Surveys, vol. 12, nº. 4, págs. 403-417, diciembre 1980.

[3] J. R. Bitner y E. M. Reingold; *"Backtrack programming techniques"*, Communications of the ACM, vol. 18, nº. 11, págs. 651-656, noviembre 1975.

[4] G. Brassard y P. Bratley; *"Algorithmics: Theory and Practice"*, Prentice-Hall, 1988. Traduc. al español de R. Peña Marí: *Algorítmica: concepción y análisis*, Masson, 1992.

[5] T. H. Cormen, C. E. Leiserson y R. L. Rivest; *"Introduction to Algorithms"*, The MIT Press, 1990.

[6] J. Galve Francés, J. C. González Moreno, A. Sánchez Calle y J. A. Velázquez Iturbide; *"Algorítmica: diseño y análisis de algoritmos funcionales e imperativos"*, Ra-Ma, 1993.

[7] E. Horowitz y S. Sahni; *"Fundamentals of Computer Algorithms"*, Computer Science Press, 1978.

[8] E. Horowitz y S. Sahni; *"Fundamentals of Data Structures in Pascal"*, Computer Science Press, 1990.

[9] T. C. Hu; *"Combinatorial Algorithms"*, Addison-Wesley, 1982.

[10] U. Manber; *"Introduction to Algorithms: A Creative Approach"*, Addison-Wesley, 1989.

[11] R. Peña Marí; *"El esquema de programación dinámica"*, informe técnico RT86/06, junio 1986.

[12] A. B. Tucker (comp.); *"Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force"*, ACM Press, 1991.

[13] J. A. Velázquez Iturbide; *"Un ejemplo clásico de programación dinámica: el problema de la mochila 0/1"*, informe técnico FIM/89.1/LSIIS/95, Facultad de Informática, Universidad Politécnica de Madrid, septiembre 1995.

[14] N. Wirth; *"Algorithms + Data Structures = Programs"*, Prentice-Hall, 1976. Traducción al español de A. Álvarez Rodríguez y J. Cuena Bartolomé: *"Algoritmos + estructuras de datos = programas"*, Ed. Castillo, 1980.

[15] W. A. Wulf, M. Shaw, P. N. Hilfinger y L. Flon; *"Fundamental Structures of Computer Science"*, Addison-Wesley, 1981.

Informática Gráfica

Sobremuestreo y Paralelización; dos formas de mejorar el Trazado de Rayos

Emilio Camahort*, Enrique S. Quintana**, Gregorio Quintana***
*Dpt. of Computer Sciences, University of Texas at Austin
E-mail: ecamahor@cs.utexas.edu
** Dpto. de Sistemas Informáticos y Computación, UPV
E-mail: equintan@dsic.upv.es
***Dpto. de Sistemas Informáticos y Computación, UPV
E-mail: gquintan@dsic.upv.es

Resumen: *una de las áreas más importantes de la informática gráfica es la visualización en tres dimensiones. Entre las técnicas de visualización 3-D destaca el método de trazado de rayos por el realismo en las imágenes que genera. Este método, sin embargo, plantea dos problemas: el excesivo tiempo de ejecución que requiere para generar una imagen y el efecto de enmascaramiento (aliasing) de altas frecuencias en la imagen, más comúnmente conocido como efecto escalera.*

En este artículo presentamos varios métodos de trazado de rayos que permiten soslayar en gran medida estos dos problemas. Para acelerar la ejecución de los métodos hemos diseñado un algoritmo paralelo de trazado de rayos que hemos codificado sobre un multiprocesador de memoria compartida Alliant FX/80. Para resolver los problemas de enmascaramiento hemos incorporado al algoritmo técnicas de sobremuestreo, pues son las más adecuadas para trazado de rayos. Concretamente, hemos utilizado las técnicas de sobremuestreo uniforme, adaptativo, estadístico y estocástico.

Con este objetivo hemos tomado como punto de partida un trazador de rayos secuencial escrito en C, que permite visualizar imágenes de escenas construidas mediante árboles CSG, primitivas sólidas sencillas y superficies poligonales. A continuación hemos codificado la parte concurrente de la aplicación en el lenguaje de programación FX/Ada. Los programas resultantes permiten reducir sustancialmente los tiempos de ejecución y minimizar el número de visuales a trazar durante el proceso de visualización. En este artículo describimos el proceso de diseño de estos algoritmos y presentamos algunas imágenes generadas por los programas.

1. Introducción

Durante los últimos diez años los gráficos por ordenador han pasado de ser una disciplina secundaria de la informática a formar parte de la mayoría de aplicaciones informáticas que requieren interacción con el usuario. Este auge de los gráficos por ordenador se ha visto impulsado también por su utilidad en la representación de imágenes que, por diversos motivos, son difíciles de obtener mediante métodos tradicionales. En este sentido podemos distinguir diferentes campos de aplicación, como el diseño asistido por ordenador, los videojuegos interactivos, la visualización científica, la generación de imágenes tridimensiones, la simulación de entornos complejos y la animación por ordenador.

En este artículo nos concentramos en una clase de métodos para la generación de imágenes tridimensionales: los métodos de trazado de rayos o, sencillamente, el *trazado de rayos*. Estos métodos han sido utilizados para CAD, visualización, simulación y animación por ordenador debido a la precisión con que modelan determinados efectos de iluminación, concretamente las sombras, la reflexión especular y la transparencia. Los dos problemas más importantes del trazado de rayos son su elevado tiempo de ejecución y el efecto de enmascaramiento de altas frecuencias, efecto que también aparece en otros métodos de generación de imágenes por ordenador.

Para reducir los tiempos de ejecución del trazado de rayos se han ideado multitud de técnicas basadas en descomposición espacial y en volúmenes de inclusión jerárquicos. Además de utilizar algunos de estos métodos, nosotros presentamos un esquema de paralelización del trazado de rayos para multiprocesadores con memoria compartida que permite reducir aún más los tiempos de ejecución del método. Para resolver el problema del enmascaramiento o *aliasing* de altas frecuencias existen en la bibliografía distintas soluciones basadas en técnicas de sobremuestreo. En nuestro trabajo hemos incorporado las más importantes al algoritmo paralelo de trazado de rayos para obtener una aplicación que permite generar imágenes tridimensionales de gran calidad en un tiempo razonable.

El resto de este artículo está estructurado de la siguiente forma. En los dos apartados siguientes se introducen los conceptos más importantes relacionados con la generación de imágenes tridimensionales y el trazado de rayos. En el cuarto apartado se describen el efecto del enmascaramiento y los métodos de sobremuestreo ideados para reducirlo. A continuación se presentan el modelo de computación paralela utilizado en nuestro diseño y los detalles de implementación sobre la máquina destino, un Alliant FX/80. Por último, en los apartados finales presentamos los resultados, imágenes y conclusiones de nuestro trabajo.

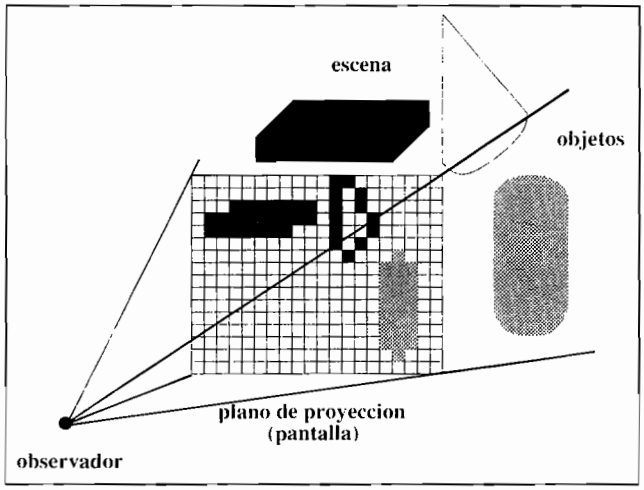


Figura 1: Generación de una imagen tridimensional

2. Precedentes

La generación por ordenador de imágenes tridimensionales consiste en representar en la pantalla del ordenador una imagen de una escena a partir de su descripción en tres dimensiones. Para representar la imagen se proyectan los objetos de la escena sobre un plano de proyección, que corresponde a la pantalla del ordenador, de forma similar a como se proyecta una escena sobre la película de una cámara fotográfica. Este proceso se ilustra en la **figura 1**.

La escena se almacena en una base de datos que contiene una descripción de los objetos en términos de su geometría y de las características de su superficie. Entre estas últimas se incluye su normal, color, transparencia, rugosidad, etc. Además de los objetos de la escena se necesitan: (i) fuentes de luz que iluminen los objetos y (ii) un modelo de cámara que represente los parámetros de posición, orientación, campo de visión, etc. del observador. A través de este modelo de cámara se proyecta la escena 3-D sobre un plano 2-D, que coincide con la pantalla del ordenador, y se determinan la cantidad y color de la luz que llega a cada pixel de la pantalla a partir de los datos de las fuentes de luz y las características de las superficies de la escena.

En el resto de este artículo llamaremos *cálculo de la visibilidad* o, simplemente, visibilidad al proceso de representar una escena 3-D sobre la pantalla 2-D de un ordenador. Una definición alternativa de visibilidad es la obtención de los objetos de la escena visibles al observador a través del modelo de cámara y el cálculo de la intensidad luminosa que, partiendo de ellos, llega hasta el observador. Debe notarse que en algún momento entre la proyección de la escena y la coloración de los pixels se produce un proceso de digitalización 2-D, debido a que la imagen proyectada es convertida en una imagen digital para representarla en la pantalla del ordenador. En adelante llamaremos a este proceso rasterización o conversión al ráster, pues ráster es el nombre de la memoria de la pantalla del ordenador.

Existen dos tipos de algoritmos de visibilidad: (i) algoritmos de proyección y (ii) algoritmos de trazado de rayos. Los algoritmos de **proyección** consisten en proyectar explícitamente los objetos de la escena, ya sea antes o después de determinar si son visibles al observador. Los algoritmos de **trazado** de rayos proyectan los objetos implícitamente, lanzando rayos que parten del observador y atraviesan cada uno de los pixels de la imagen. La visibilidad se determina intersectando cada rayo con los objetos de la escena y determinando qué punto de intersección se encuentra más próximo al observador (**figura 2**).

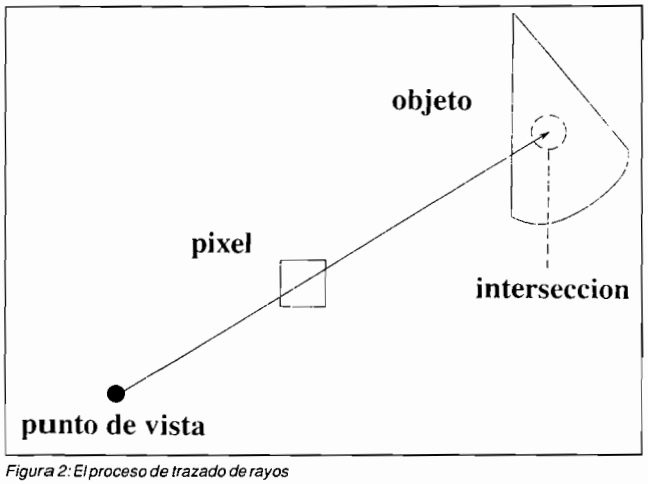


Figura 2: El proceso de trazado de rayos

Existen numerosas publicaciones sobre visibilidad en 3-D. Un buen punto de partida es el libro de Foley, van Dam, Hughes y Feiner [5], que está considerado como la más completa referencia sobre informática gráfica. Para estudiar métodos de iluminación y sombreado, que permiten determinar el color e intensidad de los pixels, recomendamos el libro de Roy Hall [9] y, a un nivel distinto, el de Cohen y Wallace [3], que se centra en radiosidad. Según el tipo de algoritmo de visibilidad que se desee utilizar existen varias referencias que tratan cada uno de ellos. Para algoritmos de proyección los libros de Joy, Grant, Max y Hatfield [10] y Rogers [12] incluyen descripciones detalladas de los más importantes, mientras que el libro de Glassner [6] contiene una descripción completa de las distintas variantes de trazado de rayos.

3. Trazado de Rayos

El método de trazado de rayos fue propuesto en 1968 y se trata del método más sencillo, intuitivo y versátil para calcular la visibilidad de una escena. Sin embargo, su elevado coste temporal motivó que no empezara a extenderse hasta 1980 [14]. La idea del método de trazado de rayos consiste en lanzar rayos que, partiendo del observador, atraviesan un determinado punto del plano de la imagen y se dirigen hacia la escena. A continuación, cada rayo se intersecta con los objetos de la escena para determinar el objeto más cercano al observador a partir de la distancia del punto de intersección al origen del rayo. Finalmente, se obtienen los datos de iluminación de la escena y del objeto más cercano y se determinan la intensidad y el color del objeto en el punto de intersección aplicando una función de sombreado. El resultado de esta función es la intensidad que llega al observador a través del rayo y que denominaremos en adelante muestra de la imagen. Los rayos que, partiendo del observador, corresponden a las muestras de la imagen se denominan rayos primarios o visuales.

El concepto básico de trazado de rayos es muy fácil de entender y, por tanto, tiene múltiples ventajas. La primera es que permite utilizar distintas representaciones para los objetos de la escena: polígonos, primitivas geométricas, superficies libres, geometría sólida constructiva y un largo etcétera. En general, un trazador de rayos puede visualizar cualquier representación geométrica que: (i) disponga de un algoritmo para calcular su intersección con una recta y (ii) permita obtener una estimación de la normal a su superficie en el punto de intersección con la recta. La normal es necesaria pues la mayoría de funciones de sombreado la utilizan como uno de sus argumentos.

Otra ventaja del trazado de rayos es su versatilidad para simular efectos de iluminación mediante el trazado de rayos que parten de los puntos de intersección con los objetos visibles al observador. El ejemplo más sencillo es la simulación de sombras arrojadas. Para ello se lanzan rayos desde el punto de intersección en dirección a cada una de las fuentes de luz

de la escena y se determina si los rayos intersectan algún objeto antes de alcanzar la fuente de luz. En caso negativo, la fuente de luz se utiliza para calcular la intensidad en el origen del rayo; si no, la función de sombreado no considera la fuente de luz. Esta técnica se denomina trazado de rayos recursivo y permite simular efectos de reflexión especular y transparencia, además de sombras arrojadas [6] [9]. Los rayos utilizados para simular estos efectos se denominan rayos secundarios y, agrupados por pixels, forman una estructura de árbol llamada árbol de rayos de un pixel, donde la visual es la raíz del árbol, los rayos de reflexión y transmisión son los nodos intermedios y los rayos de sombras arrojadas son las hojas.

El principal inconveniente del trazado de rayos es el elevado tiempo de computación que requiere el cálculo de intersecciones rayo-superficie, típicamente entre un 75% y un 95% del tiempo total de cómputo del algoritmo. Concretamente, el trazado de rayos es entre seis y siete veces más lento que el método de proyección llamado Z-buffer [12], que además es fácilmente implementable físicamente en un procesador gráfico. Para soslayar el problema del cálculo de intersecciones se han propuesto varias técnicas agrupadas en dos clases: (i) volúmenes de inclusión y (ii) enumeración espacial [6]. En este artículo nos limitamos a describir la primera de estas técnicas, que consiste en rodear cada objeto de la escena mediante una esfera (llamada bola) o un paralelepípedo (llamado caja) de forma que la intersección entre un rayo y una bola o una caja requiere sustancialmente menos tiempo que calcular la intersección entre el rayo y el objeto en sí. Si el rayo no atraviesa la bola o caja, entonces no es necesario calcular la intersección con el objeto, pues éste se encuentra dentro de la bola o caja. En la práctica esta técnica reduce el tiempo de ejecución del trazado de rayos entre dos y tres órdenes de magnitud.

Por otro lado es bastante usual agrupar los volúmenes de inclusión y construir una estructura jerárquica donde los nodos intermedios son volúmenes de inclusión y las hojas son objetos de la escena, de forma que sólo un número pequeño de intersecciones rayo-objeto es necesario para cada rayo. Existen numerosas variantes de esta técnica; el lector puede consultar el artículo de Weghorst, Hooper y Greenberg [13] como introducción al empleo de los volúmenes de inclusión y el libro de Glassner [6] como referencia completa de sus distintas variantes.

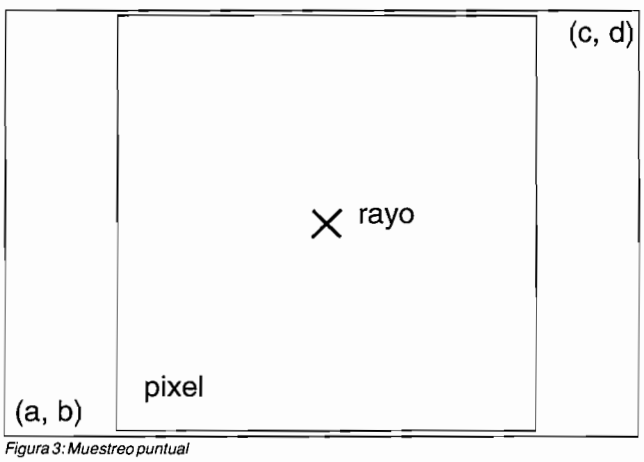


Figura 3: Muestreo puntual

Finalmente, debemos añadir que para obtener la imagen se asigna a cada pixel de la pantalla el valor de la muestra obtenida mediante un rayo trazado a través del centro del pixel, tal y como se muestra en la figura 2. Esta aproximación, que asume que la intensidad de todo el pixel es igual a la intensidad en su centro, se denomina muestreo puntual, porque para determinar la intensidad de un pixel se toma una muestra en un punto del plano 2D de la imagen y se colorea el área del pixel con el valor de esa única muestra. En la figura 3 aparece una representación gráfica de este método, donde el recuadro indica el área del pixel y el aspa el punto a través del cual se traza el rayo en dirección hacia la escena.

4. Enmascaramiento y Sobremuestreo

El problema de la técnica de muestreo puntual es el enmascaramiento de las altas frecuencias espaciales de la imagen (*aliasing*), también conocido como *efecto escalera*. Este efecto, común a la mayoría de problemas de representación gráfica, se produce al tomar la imagen proyectada original (**figura 4(a)**) y discretizarla antes de almacenarla en el ráster (**figura 4(b)**). Durante este proceso de discretización parte de la información de la imagen se pierde, concretamente las zonas de la imagen más afectadas son aquellas donde se producen los cambios más bruscos de intensidad. En estas zonas aparecen frecuencias espaciales por encima del límite de Shannon-Nyquist establecido por la resolución de la pantalla del ordenador. El lector interesado en los problemas de muestreo y reconstrucción de imágenes pueden consultar el libro sobre tratamiento digital de imágenes de González y Woods [7].

Desde un punto de vista más intuitivo este problema se produce debido a que la pantalla del ordenador se divide en pixels con una determinada área. La solución ideal para resolverlo consiste en colorear el pixel con la media de la intensidad $I(x,y)$ de la imagen dentro del área cubierta por el pixel (figura 4(c)). Supongamos que el pixel tiene forma rectangular y que sus vértices tienen coordenadas (a,b) y (c,d), respectivamente (figura 3). Si llamamos al color del pixel $C(a,b)$ entonces su valor ideal es la esperanza de $I(x,y)$ entre los límites del pixel:

$$C(a,b) = \int_a^c \int_b^d x\, y\, I(x,y)\, dy\, dx$$

Sin embargo, en la mayoría de casos prácticos I(x,y) no es conocida y, por tanto, es necesario aproximar el valor de esta integral. Para ello se han propuesto varias técnicas que dependen del tipo de algoritmo de visualización utilizado. La aproximación más sencilla consiste en obtener la intensidad que llega a un punto (x,y) dentro del pixel y asumir que ésta es la intensidad del pixel:

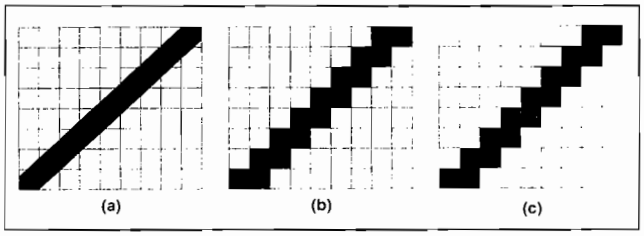


Figura 4: (a) Imagen original (sin rasterizar), (b) enmascaramiento (aliasing) o efecto escalera y (c) desenmascaramiento (antialiasing)

$C(a,b)$ $I(x,y)$ donde x $[a,c]$ y $[b,d]$

Típicamente el punto (x,y) que se elige es el centro geométrico del pixel

$$(x,y) = (\tfrac{a+c}{2}, \tfrac{b+d}{2}).$$

Tal es el caso de la aproximación utilizada por el muestreo puntual para trazado de rayos. Para algoritmos de proyección existen otras técnicas que se agrupan bajo el nombre de técnicas de *desenmascaramiento* (o *antialiasing*) y que se detallan en los libros de Foley *et al.* [5] y Joy *et al.* [10].

En el resto de este artículo nos concentramos en técnicas de desenmascaramiento para trazado de rayos o técnicas de *sobremuestreo*, que consisten en trazar rayos adicionales a través del pixel y obtener una estimación del color del pixel a partir de la media ponderada de las muestras obtenidas:

$$C(a,b) \approx \sum_x \sum_y \frac{A(x,y)}{A} I(x,y)$$

donde $A = (c\bar{a})(d\bar{b})$ es el área del pixel y $A(x,y)$ es el área de la porción del pixel que corresponde al punto (x,y). De acuerdo con la forma de elegir los puntos del pixel a muestrear, distinguimos cuatro técnicas de sobremuestreo.

Sobremuestreo Uniforme

Dado un número de muestras determinado a priori, esta técnica distribuye dichas muestras uniformemente sobre la superficie de cada pixel. El número de muestras y su posición son los mismos para todos los pixels y determinan la relación entre la calidad de la imagen y el tiempo necesario para su generación. En nuestro caso hemos elegido 22, 33 y 44 muestras por pixel (**figura 5**). **La figura 6** tiene un diagrama con los pesos $A(x,y)/A$ correspondientes a estas resoluciones de sobremuestreo.

Sobremuestreo Adaptativo

Este método aprovecha la característica de *coherencia espacial* de las imágenes: 'es muy probable que la variación de color entre dos muestras de un mismo pixel sea pequeña'. Por tanto, en el sobremuestreo adaptativo se obtienen más muestras de aquellas regiones de la imagen donde se producen mayores cambios de intensidad. Una forma de aplicar este método consiste en trazar inicialmente cinco visuales a través del pixel y comparar sus intensidades dos a dos. Si existe una diferencia sustancial mayor que un valor umbral entre dos de ellas, entonces se toma el subpixel situado entre esos dos rayos y se trazan más visuales de acuerdo con el mismo algoritmo. Este proceso se repite de forma recursiva hasta que la diferencia entre intensidades es menor que una cota preestablecida o el área de los subpixels resulta demasiado pequeña.

Para limitar el proceso recursivo de subdivisión de pixel hemos utilizado en nuestros experimentos una cota igual a un paso de resolución de color, debido a que la escala disponible se limita a 256 colores. También hemos limitado el nivel de recursividad a un máximo de tres subdivisiones, ya que el peso de una muestra por debajo de este nivel es insignificante respecto a la intensidad total del pixel. Por ejemplo, en la **figura 7(a)** la esquina inferior derecha del pixel tiene un cambio brusco de intensidad, por tanto, sólo es necesario obtener muestras adicionales del cuadrante inferior derecho del pixel y de cada uno de sus subpixels (**figura 7(b)**). Los pesos en este caso se calculan acumulando los pesos relativos de cada muestra en cada uno de los pasos de subdivisión (**figura 7(c)**).

Sobremuestreo Estadístico

Los métodos anteriores permiten reducir algunos efectos debidos al enmascaramiento, pero no resuelven el problema en general, pues siempre es posible encontrar imágenes en las que el efecto escalera se puede observar, incluso cuando el número de muestras por pixel es elevado. El motivo es que los patrones de muestreo de estos métodos son regulares y los efectos que producen son fácilmente detectables por el ojo humano. Por este motivo, en el sobremuestreo estadístico las muestras se distribuyen aleatoriamente sobre la superficie del pixel, de forma que los errores introducidos en la imagen generada son ruido blanco de media cero y, por tanto, más difíciles de identificar por el ojo humano.

De acuerdo con la técnica de Lee, Redner y Uselton [11], este método utiliza un número variable de muestras por pixel, que depende de un test estadístico que indica cuando la aproximación de la intensidad del pixel está suficientemente cerca del valor real. Para el análisis de este método, consideramos un conjunto infinito de muestras distribuidas normalmente sobre la superficie del pixel con media $\bar{x}$ y varianza σ^2 desconocidas. El objetivo es obtener, a partir de un número de muestras n, un estimador de la media $\bar{x}_n$ suficientemente próximo al valor real $\bar{x} = C(a,b)$. Lee, Redner y Uselton afirman que el error en la estimación de $\bar{x}$ se puede acotar utilizando la distribución t de Student de la siguiente forma:

$$|\bar{x}_n - x| < t\frac{s_n}{\sqrt{n}}$$

donde s_n es una estimación de la desviación típica. La idea pues consiste en elegir un intervalo de confianza para la media $\bar{x}$ y a continuación tomar muestras y actualizar el valor de los

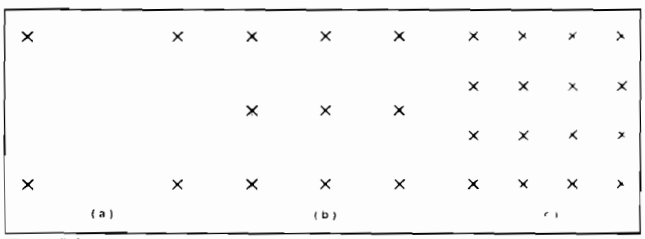


Figura 5: Sobremuestreo uniforme: (a) 4 muestras por pixel, (b) 9 muestras por pixel y (c) 16 muestras por pixel.

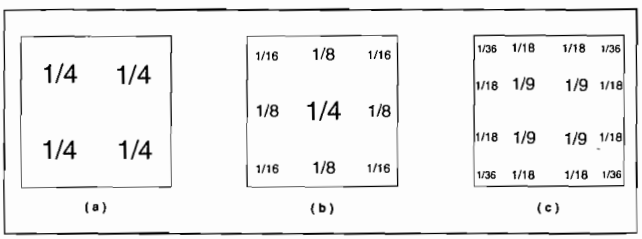


Figura 6: Asignación de pesos a las muestras del algoritmo de sobremuestreo uniforme: (a) 4 muestras por pixel, (b) 9 muestras por pixel y (c) 16 muestras por pixel.

estimadores $\bar{x}_n$ y s_n consecuentemente. El cálculo de la intensidad concluye cuando la desigualdad anterior es satisfecha. P.ej. si queremos aproximar el valor de $\bar{x}$ dentro de un intervalo de confianza del 95% y con un riesgo de primera especie (riesgo de equivocarse) del 5%, entonces al final del cómputo de cada muestra debemos comprobar si n, $\bar{x}_n$ y s_n satisfacen:

$$\frac{t\, s_n}{\bar{x}_n \sqrt{n}} < 0.05$$

en cuyo caso el valor final del pixel $\bar{x}_n$ estará en el intervalo $[0.95\,\bar{x}\,;\,1.05\,\bar{x}\,]$ con un 95% de probabilidad. La **figura 8(a)** muestra un posible conjunto de muestras de un pixel adquiridas mediante este método para estimar su intensidad media.

Sobremuestreo Estocástico

Por último, el sobremuestro estocástico introducido por Cook [4] utiliza una técnica híbrida llamada *jittered sampling* (muestreo *nervioso* o *tembloroso*), que combina las ventajas del sobremuestreo uniforme y del sobremuestreo estadístico. En este método, las muestras se toman según patrones casi uniformes, análogos a los empleados en el sobremuestreo uniforme, pero con las coordenadas de cada muestra perturbadas ligeramente (**figura 8(b)**). Además, después de obtener el valor de cada muestra se añade a su intensidad una pequeña perturbación aleatoria para, finalmente, asignar al valor del pixel la media aritmética de todas las muestras tomadas en su interior. Nótese que en esta técnica los pesos asignados a cada muestra son los mismos. Para nuestra realización hemos utilizado esta técnica con 1, 4, 9 y 16 muestras por pixel y una perturbación aleatoria de ±5%.

En su artículo Cook justifica con detalle las ventajas de esta técnica frente al resto de métodos de sobremuestreo, pero su análisis tiene un elevado contenido técnico y hemos preferido omitirlo en este artículo. Los lectores interesados en él pueden consultar el libro de Foley *et al.* [5] o el de Joy *et al.* [10], además del artículo original de Cook [4].

5. Paralelización

El principal inconveniente de los métodos de sobremuestreo es el incremento de los tiempos de ejecución que su empleo supone. P.ej. las técnicas de sobremuestreo uniforme y estocástico con nn rayos por pixel requieren respectivamente $(n1)^2$ y n^2 veces el tiempo que consume la técnica de muestreo puntual. Del mismo modo, el muestreo adaptativo es al menos dos veces más lento que el muestreo puntual y el muestreo estadístico requiere un mínimo de n_0 veces el tiempo del muestreo puntual, donde n_0 es el número inicial de muestras por pixel.

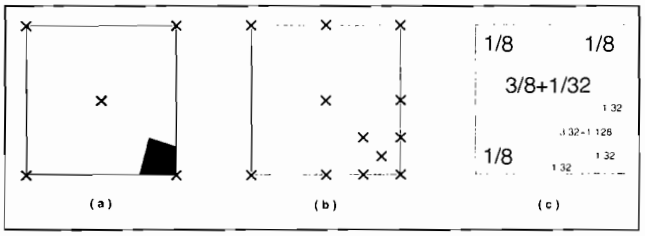


Figura 7: Algoritmo de sobremuestreo adaptativo: (a) ejemplo, (b) subdivisión del pixel y (c) cálculo de pesos

Una forma de soslayar este problema consiste en extender los algoritmos para ejecutarlos sobre una arquitectura paralela. Esta extensión viene justificada por la facilidad de paralelizar los algoritmos de trazado de rayos y por la disponibilidad en el mercado de máquinas paralelas cada vez más potentes y de coste más reducido. Por ello en este artículo nos concentramos en adaptar los métodos anteriores a una máquina de propósito general con arquitectura multiprocesador y memoria compartida. Los libros de Glassner [6] y Green [8] contienen información sobre algoritmos paralelos de trazado de rayos para otros tipos de arquitecturas. Nuestro objetivo es obtener un visualizador paralelo que incluya distintos métodos de sobremuestreo para generar eficientemente imágenes de calidad. Los métodos paralelos que presentamos son una extensión de un trazador de rayos secuencial desarrollado por Camahort [1] y paralelizado por Camahort, Quintana, Vivó y Vidal [2].

Inicialmente asumimos un esquema de paralelización formado por un conjunto de tareas o procesos concurrentes que se ejecutan independientemente sobre cada procesador y que colaboran entre sí para la generación de la imagen. Considerando el tipo de arquitectura que vamos a utilizar podemos elegir entre varios esquemas de paralelización de acuerdo con dos criterios de selección:

(i) El nivel de granularidad de los algoritmos: podemos elegir entre una realización de grano fino, que consiste en paralelizar procesos sencillos, como el cálculo de intersecciones rayo-superficie, o una realización de grano grueso que asigna a los procesadores tareas más complejas, como el trazado de un rayo, la obtención de una muestra o la iluminación de un pixel. De estas dos alternativas, la primera requiere un elevado número de sincronizaciones que afecta sustancialmente a la eficiencia de los algoritmos. La segunda, sin embargo, requiere un número de sincronizaciones menor y permite utilizar una estrategia de *pool* de procesos, donde un proceso se encarga de asignar trabajo a los demás y el resto de procesos sólo se dedican a llevar a cabo el trabajo encomendado. Por este motivo elegimos una realización de grano grueso para seguidamente seleccionar la unidad de asignación de trabajo a cada tarea: un rayo, una visual (o muestra) o un pixel.

(ii) la unidad de trabajo asignada a cada tarea. Asignar un rayo a cada tarea es el esquema menos adecuado, debido a que

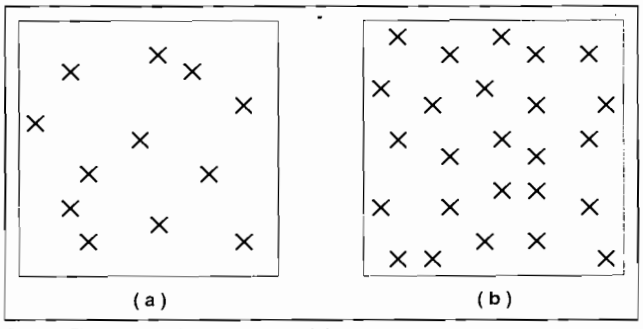


Figura 8: Ejemplos: (a) sobremuestreo estadístico y (b) sobremuestreo estocástico

la recursividad del método incrementa demasiado el coste de sincronización en el árbol de rayos de un pixel. En cuanto a tratar un pixel en cada procesador, es una solución interesante para el método de muestreo puntual, pero resulta ineficiente para los de sobremuestreo, pues éstos utilizan cada muestra para calcular la intensidad de varios pixels y su empleo resultaría en dos o más tareas trazando la misma visual. En consecuencia utilizamos un esquema de paralelización orientado a muestras o visuales de forma que mediante mecanismos de sincronización es posible garantizar una ocupación máxima de los procesadores y evitar la duplicidad de tareas de trazado.

La **figura 9** muestra un diagrama con los bloques más relevantes del algoritmo paralelo, donde la tarea de sincronización es la que, al iniciarse el proceso de visualización, obtiene los datos de la imagen y genera un conjunto de visuales a trazar de acuerdo con el algoritmo de muestreo seleccionado para la ejecución. Estas visuales se almacenan en una bolsa de visuales pendientes que administra la tarea de sincronización. A continuación las tareas de trazado solicitan visuales para procesar, que son extraídas y asignadas a cada una de ellas por la tarea de sincronización. Entonces las tareas de trazado procesan las visuales independientemente, es decir calculan la intensidad que llega al observador a través de ellas, y devuelven su resultado, una muestra de la imagen, a la tarea de sincronización. A partir de las muestras obtenidas la tarea de sincronización actualiza la información que posee sobre los pixels de la imagen, concretamente el color de los pixels afectados por cada muestra. En el caso de sobremuestreo uniforme los valores obtenidos deben ponderarse de acuerdo con la porción de pixel afectada. En los métodos adaptativo y estadístico el cálculo de una determinada visual puede provocar la inserción de nuevas visuales en la bolsa de visuales.

Este proceso se repite hasta que todos los valores de los pixels han sido calculados y la bolsa de visuales se vacía. Debe notarse que la relación entre visuales y pixels es, en el caso más general, una relación *muchos-a-muchos*, cuya consistencia debe ser mantenida por la tarea de sincronización. De ahí que el acceso a las estructuras de datos correspondientes sólo se lleve a cabo a través de esta tarea. Aunque pueda parecer que esta estrategia provoca un cuello de botella en la tarea de sincronización, la realidad es que el proceso de una visual es mucho más lento que las operaciones de sincronización y, por tanto, las posibilidades de contención a ese nivel son mínimas.

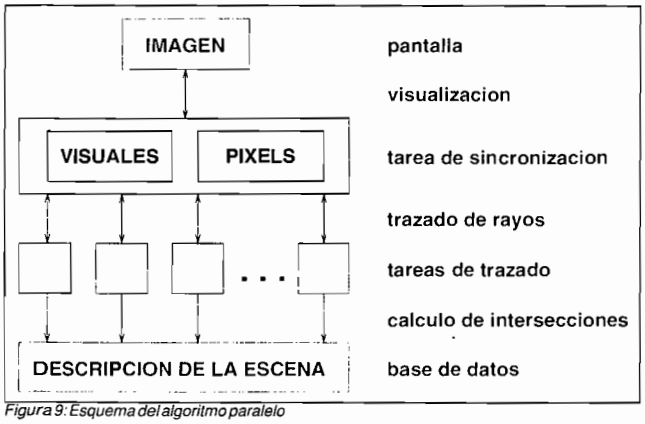


Figura 9: Esquema del algoritmo paralelo

6. Implementación

Para probar los algoritmos descritos anteriormente hemos realizado una implementación sobre un multiprocesador Alliant FX/80 con memoria compartida. Este multiprocesador soporta hasta ocho elementos de computación (*CE's*) y doce procesadores interactivos (*IP's*) dedicados, respectivamente, a tareas de proceso y tareas auxiliares. Por tareas auxiliares entendemos la ejecución de procesos del sistema operativo, la gestión de entrada/salida y la gestión procesos que no requieren computación numérica. La memoria del sistema consta de tres niveles: (i) memoria virtual de 2 Gbytes, (ii) memoria central de 256 Mbytes, y (iii) dos tipos de antememorias de 512 Kbytes para los *CE's* y 128 Kbytes para los *IP's* como máximo. El acceso a la memoria central se efectúa a través de un bus común a todos los procesadores.

El trazador implementado permite visualizar árboles CSG, modelos de fronteras poligonales y ciertos tipos de superficies libres a través de un modelo de cámara puntual. La función de sombreado es configurable, está basada en el modelo de iluminación de Hall [9] y permite utilizar distintas clases de fuentes de luz, incluidas fuentes de luz distribuidas. El programa acepta como entrada una descripción jerárquica de los objetos de la escena y utiliza volúmenes de inclusión, tanto bolas como cajas, para limitar el número de operaciones de intersección rayo-superficie.

Las partes del trazador dedicadas a la gestión de los datos de la escena, el cálculo de intersecciones, la función de sombreado y el manejo de la imagen se han implementado en C y proceden de una versión secuencial anterior, cuyos detalles están descritos en [1]. El resto del código, dedicado al sistema de tareas y a los algoritmos de sobremuestreo, está escrito en FX/Ada, el lenguaje de programación paralela soportado por el Alliant FX/80. Cada tarea del algoritmo original se ha codificado como una tarea en Ada y el número de tareas de trazado se ha dejado como un parámetro configurable por el usuario de acuerdo con el número de procesadores disponibles en el sistema.

La estructura del programa corresponde a la del algoritmo paralelo de la figura 9, donde los módulos sombreados reflejan la parte paralela del código implementada en FX/Ada. Nótese que una parte de cada tarea de trazado es paralela y la otra parte, concretamente la obtención de una muestra de la imagen, es secuencial de acuerdo con nuestros criterios de implementación. La descomposición en módulos de los bloques de la figura 9, así como otros detalles de implementación del sistema los omitimos en este artículo, pero se pueden encontrar en las referencias [1] y [2].

7. Resultados

Una vez completada la versión paralela del trazador de rayos hemos generado varias imágenes de 400320 pixels a partir de cinco escenas diferentes. Para analizar los resultados hemos medido los tiempos de ejecución y hemos calculado los incrementos de velocidad (*speedups*) respecto a la versión secuencial, dividiendo el tiempo de cómputo de la versión secuencial entre el tiempo de cómputo de la versión paralela.

Los resultados obtenidos indican que las versiones secuencial y paralela sobre un procesador difieren muy poco en tiempo de ejecución, tan sólo un 1%. Esto significa que la sobrecarga del sistema (*overhead*) producida por el esquema de tareas en Ada es despreciable y los incrementos de velocidad obtenidos pueden considerarse buenos respecto a la versión puramente secuencial en C. En adelante, pues, los resultados presentados proceden de ejecuciones de la versión de ocho tareas sobre uno y ocho procesadores y los incrementos de velocidad se obtienen respecto a la versión paralela sobre un procesador.

En la **tabla 2** mostramos los tiempos medios de ejecución en segundos, los incrementos medios de velocidad y la eficiencia para las distintas técnicas de sobremuestreo empleadas. Nótese que el *speedup* teórico es 8 disponemos de 8 *CE's* mientras que los *speedups* obtenidos se sitúan alrededor de 7, es decir, la eficiencia de los métodos supera en la mayoría de los casos el 85%. Si comparamos estos resultados con los obtenidos mediante aplicaciones de manejo de matrices y de resolución de problemas de algebra lineal, podemos afirmar que los resultados son muy buenos, pues estas aplicaciones suelen alcanzar unos niveles de eficiencia similares a los obtenidos por nuestro trazador de rayos. Otro resultado interesante es el hecho de que el sistema de tareas en Ada y la paralelización de los algoritmos de sobremuestreo sólo reducen la eficiencia en un 10% aproximadamente. Esto es sin aprovechar la capacidad de procesamiento vectorial del Alliant FX/80 debido a que en este tipo de aplicaciones los vectores son pequeños y el cálculo de intersecciones es ineficiente y difícil de vectorizar.

Las **figuras 10 a 17** (ver los interiores de la portada y contraportada) muestran varias imágenes generadas con el trazador de rayos paralelo. Las **figuras 10 a 13** permiten comparar el efecto producido por las distintas técnicas de muestreo al haber sido generadas con una resolución de 4032 pixels y ampliadas posteriormente a una resolución 256 veces mayor: 640512 pixels. La **imagen 10**, generada con muestreo puntual, permite observar el efecto de enmascaramiento sobre un objeto peque-ño. Las imágenes siguientes (**figuras 11, 12 y 13**) representan el método de sobremuestreo uniforme con 4, 9 y 16 rayos por pixel, respectivamente. Las mayores diferencias se pueden observar en el contorno de las figuras donde la variación de inten-sidad de los pixels es más o menos suave en función de la precisión del método. No hemos incluido imágenes obtenidas mediante los métodos de sobremuestreo adaptativo y estadístico, debido a que no presentan diferencia alguna respecto a la **imagen 13**. La escena completa con una resolución de 12801024 se representa en la **imagen 14**. Las **imágenes 15 y 16** están generadas con muestreo puntual y 4 rayos por pixel, respectivamente. Por último, la **imagen 17** se obtuvo con un muestreo uniforme de 9 rayos por pixel.

8. Conclusiones

En este artículo hemos introducido los conceptos más importantes relacionados con la generación de imágenes por ordenador a partir de modelos de escenas tridimensionales. Entre ellos hemos comentado el modelado de objetos, el cálculo de la visibilidad, la iluminación y el sombreado, y la conversión al ráster.

También hemos hecho breve referencia a los problemas de enmascaramiento y a diferentes técnicas de visualización para finalmente concentrarnos en el método de visibilidad de trazado de rayos. Las características más relevantes de este método son su simplicidad conceptual y su versatilidad que permite simular fácilmente efectos de múltiples fuentes de luz, sombras arrojadas, reflexión especular y transparencia. Su principal inconveniente es su ineficiencia temporal motivada por el elevado número de operaciones de intersección rayo-objeto que requiere. Para resolver este problema hemos apuntado dos soluciones: (i) el empleo de volúmenes de inclusión para reducir el número de intersecciones rayo-objeto necesarias y (ii) la paralelización del algoritmo, que además permite extender eficientemente las técnicas de sobremuestreo utilizadas para reducir los efectos de enmascaramiento en el trazado de rayos.

El resto del artículo se centra en la descripción de los métodos de sobremuestreo y en el diseño e implementación del algoritmo paralelo sobre un multiprocesador de propósito general y memoria compartida Alliant FX/80. La versión final del trazador de rayos consta de dos partes: una parte paralela codificada en FX/Ada con un esquema de paralelización de grano grueso basado en tareas encargadas cada una de calcular una muestra de la imagen y una parte secuencial codificada en C y encargada del resto de operaciones necesarias para el trazado de rayos. Las pruebas realizadas nos han permitido ajustar los parámetros del método, concretamente el número de tareas más adecuado, y comprobar el correcto funcionamiento de los algoritmos de sobremuestreo.

Aparte de este trabajo hemos considerado otras mejoras y ampliaciones que sería deseable añadir a la presente versión del trazador. Entre ellas destaca el empleo de otras técnicas de aceleración para el trazador de rayos secuencial (subdivisión espacial, técnicas direccionales y de coherencia) [6]. El objetivo final es realizar una comparativa de métodos de sobremuestreo para trazado de rayos a nivel de posibilidades de paralelización, eficiencia y calidad de la imagen. Por otro lado es interesante también la implementación de estos métodos sobre otros multiprocesadores distintos del utilizado para determinar qué arquitecturas son las más adecuadas para este tipo de procesos.

	secuencial	8 tareas 1 procesador	7 tareas 8 procesadores	8 tareas 8 procesadores	16 tareas 8 procesadores
tiempo medio	3356	3384	534	483	494
speedup medio	1	0,99	6,28	6,95	6,79

Tabla 1: Tiempos de ejecución y speedup del método de muestreo puntual para distintos números de tareas de trazado sobre 1 y 8 procesadores

En resumen, hemos abordado los dos problemas más importantes del método de trazado de rayos y hemos presentado dos formas de soslayarlos mediante técnicas de sobremuestreo y su adaptación a un entorno paralelo. Los resultados han sido satisfactorios, ya que se han obtenido unos excelentes incrementos de velocidad en torno a 7 para 8 procesadores que suponen un aprovechamiento de más de un 85% de la capacidad de multiproceso del Alliant FX/80. Las imágenes también corresponden a los resultados esperados tal y como se muestra en las fotografías que ilustran este artículo.

Agradecimientos

Queremos expresar nuestro agradecimiento a Bernardo Castellanos, Miguel Chover, Javier Lluch, Ricardo Quirós y Roberto Vivó por su colaboración en la realización de este trabajo y, concretamente, en la generación de las imágenes que lo acompañan. Así mismo queremos señalar que este trabajo fue realizado en parte durante el período de disfrute de dos becas de formación de personal investigador, una del Ministerio de Educación y Ciencia (Emilio Camahort) y la segunda de la *Conselleria de Cultura, Educació y Ciència* de la *Generalitat Valenciana* (Gregorio Quintana). El trabajo de Emilio Camahort ha sido también financiado por el Programa de Becas de Ampliación de Estudios en el Extranjero de la *Fundació "La Caixa"* de Barcelona. Finalmente queremos agradecer la colaboración desinteresada de Luisa Zuñiga del Departamento de Estadística de la Universidad Politécnica de Valencia.

Referencias

[1] E.Camahort; *"Diseño y Realización de un Visualizador de Arboles CSG Basado en el Modelo de Trazado de Rayos"*. Proyecto Fin de Carrera, Facultad de Informática, Valencia, oct 1990.

[2] E.Camahort, G.Quintana, R. Vivó, A.M.Vidal; *"A Parallel Implementation of a Ray Tracer on a Shared Memory Multiprocessor"*. Proceedings of ATARV'93, Ankara, Turquía, Jul 1993.

[3] M.F.Cohen, John R. Wallace; *"Radiosity and Realistic Image Synthesis"*. Academic Press, Cambridge MA, 1993.

[4] R.L.Cook; *"Stochastic Sampling in Computer Graphics"*. ACM Transactions on Graphics, 5, 1, Jan 1986, pp 51-72.

[5] J.D.Foley, A.van Dam, S.K.Feiner, J.F.Hughes; *"Computer Graphics: Principles and Practice"* (2nd Ed). Addison-Wesley, Reading MA, 1990.

[6] A.S.Glassner (Ed.); *"An Introduction to Ray Tracing"*. Academic Press, Cambridge MA, 1989

[7] R.C.Gonzalez, R.E.Woods; *"Digital Image Processing"* (3rd Ed). Addison-Wesley, Reading MA, 1992.

[8] S.Green; *"Parallel Processing for Computer Graphics"*. Pitman, Londres, 1991.

[9] R.A.Hall; *"Illumination and Color in Computer Generated Imagery"*. Springer-Verlag, New York, 1989.

[10] K.I.Joy, C.W.Grant, N.L.Max, L.Hatfield (Eds.); *"Tutorial: Computer Graphics: Image Synthesis"*. IEEE Computer Society Press, Los Alamitos CA, 1988.

[11] M.E.Lee, R.A.Redner, S.P.Uselton; *"Statistically Optimized Sampling for Distributed Ray Tracing"*. SIGGRAPH'85, Computer Graphics, 19, 3, Jul 1985, pp 61-67.

[12] D.F.Rogers; *"Procedural Elements for Computer Graphics"*. McGraw-Hill, New York, 1985.

[13] H.Weghorst, G.Hooper, D.P.Greenberg; *"Improved Computational Methods for Ray Tracing"*. ACM Transactions on Graphics, 3, 1, Jan 1984, pp 52-69.

[14] T.Whitted; *"An Improved Illumination Model for Shaded Display"*. Comms. of ACM, 23, 6, Jun 1980, pp 343-349.

		coste teórico	*coste práctico*
muestreo puntual		**1**	**1**
sobremuestreo	4 rayos/pixel	**1.057**	**1.062**
uniforme	9 rayos/pixel	**4.11**	**4.5**
	16 rayos/pixel	**9.17**	**8.80**
sobremuestreo	1 rayo /pixel	**1**	**1.02**
estocástico	4 rayos/pixel	**1.057**	**1.08**
	9 rayos/pixel	**4.11**	**3.99**
	16 rayos/pixel	**9.17**	**8.89**
sobremuestreo adaptativo		-	**3.96**
sobremuestreo estadístico		-	**3.99**

Tabla 3 comparativa entre costes teóricos y prácticos de los métodos usados

Anexo con figuras 10 a 17 en interiores de portada y contrap.

		1 procesador	8 procesadores	*speedup*	eficiencia
muestreo puntual		**3384**	**483**	**7.01**	**0.88**
sobremuestreo	**4rayos/pixels**	**3577**	**513**	**6.97**	**0.87**
uniforme	**9rayos/pixels**	**13916**	**1954**	**7.12**	**0.89**
	6rayos/pixels	**30830**	**4250**	**7.25**	**0.91**
sobremuestreo	**1rayos/pixels**	**3314**	**474**	**6.99**	**0.87**
estocástico	**4rayos/pixels**	**3516**	**522**	**6.73**	**0.84**
	9rayos/pixels	**13312**	**1928**	**6.90**	**0.86**
	6rayos/pixels	**29554**	**4294**	**6.88**	**0.86**
sobremuestreo adaptativo		**10018**	**1915**	**5.23**	**0.65**
sobremuestreo estadístico		**12848**	**1928**	**6.66**	**0.83**

Tabla 2: Tiempos de ejecución, speedup y eficiencia en función del método de muestreo utilizado

Seguridad y Sistemas

Contribuciones actuales de la gestión de la contabilidad en redes de computadores

Javier Areitio Bertolín
Universidad de Deusto (UD)
E. Mail: jareitio@orion.deusto.es
Ana Mª Areitio Bertolín
Universidad del Pais Vasco (UPV/EHU).
E. Mail: euparbea@bs.ehu.es

Resumen: *este artículo aborda la gestión de la contabilidad de redes mostrando las fases del procedimiento que lo conforma, su impacto e importancia actuales y algunos mecanismos de gestión de contabilidad clasificados y estructurados en orden a su grado/nivel de prestaciones.*

1. Objetivos

La gestión de la contabilidad de redes supone medir de forma rápida, precisa, fiable, segura, flexible y óptima la utilización que los usuarios hacen de los recursos de la red en base a establecer métricas y verificar cuotas de su utilización, permite determinar costos y facturar/tarificar a los usuarios. Supone recoger estadísticas de red de forma periódica y aperiódica, lo que puede ayudar al gestor de red a la hora de tomar decisiones y planificar a corto, medio y largo plazo sobre la actualización, distribución, y asignación de los recursos de la red de cara a conseguir una mayor calidad de red para los usuarios. Igualmente, estos datos son útiles para gestionar los recursos del sistema como espacio en disco, potencia de procesamiento, almacenamiento de respaldo (backup), etc., incluso aunque no sean explícitamente parte de la gestión de red.

2. Aportaciones principales

La gestión de la contabilidad de redes permite al gestor de red medir e informar acerca de la información de contabilidad relativa a usuarios individuales y de grupo y luego utilizar estos datos para facturar a dichos usuarios, asignar, distribuir recursos y calcular el costo por usuario de los datos que se transmiten por la red. Además, proporciona al gestor de red un conocimiento preciso de la utilización que los usuarios hacen de los recursos de red que puede ayudar a la creación de una red más eficiente, competitiva, productiva y con calidad. El facturar a los usuarios es esencial para recuperar los gastos producidos en la construcción y mantenimiento de la red de comunicación de datos. La gestión de la contabilidad también proporciona en relación a la política de facturación ayudas para ofrecer una distribución más equitativa de estos gastos. Puede ayudar a la hora de la planificación del presupuesto y del personal. Cada vez más, las organizaciones van viendo como vitales estos aspectos de la gestión de la contabilidad de red convirtiéndose en una de las cuestiones más importantes a considerar. Examinando métricas de utilización y cuotas de utilización de dichas métricas se puede asegurar que cada usuario tenga suficientes recursos para realizar las tareas que acometa. Se pueden utilizar estas estadísticas para seguir la pista de la utilización de diversos recursos de red tales como servidores de aplicaciones, servidores de computación, servidores de ficheros, servidores de impresión, servidores de comunicaciones, etc. Por ejemplo, un grupo de trabajo GT-k puede utilizar la red para acceder a un sistema de información de un servidor de aplicaciones ubicado en una organización remota. Al utilizar la información de gestión de la contabilidad, se detecta que la mayor parte del tráfico que circula por la red hacia la organización remota es del grupo de trabajo GT-k que accede a este servidor de aplicaciones. La gestión de la contabilidad proporciona la información necesaria que permite la toma de decisión de si el grupo de trabajo GT-k precisa su propio servidor de aplicaciones local o es suficiente con la conexión al servidor remoto. En un entorno de red tradicional en el que los terminales conectan con sus modems a un computador central, la gestión de la contabilidad puede ayudar a distribuir los intervalos de tiempo de compartición del computador para dicho grupo de terminales. Ciertos terminales pueden tener prioridad de utilización sobre otros. La gestión de la contabilidad puede ayudar a determinar si los usuarios utilizan, de hecho, los terminales a los que se les ha dado mayor prioridad un porcentaje máximo de tiempo o si el esquema de prioridades inicialmente establecido necesita modificarse. La gestión de la contabilidad de red puede ayudar a la hora de una distribución más racional de los diversos recursos valiosos existentes en una red. Consideremos una red local con un servidor de ficheros SF1 que mantiene actualizada información de stock importante en una base de datos. Supongamos que un usuario de un PC de la red local decide hacer una copia de respaldo (backup) de su disco completo de 210 Mbytes de su PC al SF1. El usuario comienza el 'backup' a última hora de la tarde y se marcha de la organización volviendo al día siguiente. Después de que el procedimiento de 'backup' termina, sólo queda en el servidor SF1 0,5 Mbytes de memoria libre. Horas después, el SF1 ejecuta automáticamente un programa que se encarga de recoger información de stock importante de diversos puntos remotos, sin embargo mientras se transfieren estos datos, el disco de SF1 se llena y la transferencia se detiene. A la mañana siguiente, cuando los usuarios de la red local intentan recuperar la información de stock importante del servidor SF1 se produce un mensaje de error. Para investigar el problema, se examinan las estadísticas de gestión de la contabilidad para SF1 y se averigua que un cierto usuario transferió un conjunto elevado de ficheros durante un largo período de tiempo por la noche. Ahora el gestor de red dispone de toda la información necesaria para poder actuar adecuadamente. Las técnicas de gestión de la contabilidad también pueden ayudar a las organizaciones a calcular los costos necesarios para enviar datos a través de la red para cada usuario dado, lo que permite al usuario conocer cuanto se gastó para obtener los servicios de red. Esto proporciona una distribución equitativa de los gastos asociados con el funcionamiento y mantenimiento de la red.

3. Fases del proceso de gestión de la contabilidad

La gestión de la contabilidad de redes permite al gestor de red, entre otras cosas, medir la utilización de los recursos de red en base a establecer métricas de utilización, verificar cuotas de utilización de las métricas, determinar los costos y facturar adecuadamente a los usuarios por el uso de la red. Este proceso puede dividirse en las siguientes tres fases:

1.Recogida de datos sobre utilización de los recursos de red. Para obtener la información relativa a métricas y cuotas se necesitaría recoger los datos de la gestión de la contabilidad (facturación, etc.) más o menos frecuentemente. Si los dispositivos de red pueden almacenar suficiente cantidad de datos de actividad, la recuperación periódica de los datos puede bastar..

Utilización de métricas para ayudar al establecimiento de cuotas de utilización. Las métricas pueden ayudar a conocer con qué grado de intensidad emplean los usuarios los recursos de la red. Por ejemplo, una métrica puede revelar el número de conexiones realizadas a un servidor de terminales, el número de transacciones realizadas a una base de datos concreta, o el tiempo total de 'login' (apertura de sesión) de un usuario en una cuenta de un supercomputador (Cray, Convex,...). La utilización de la información de la gestión de la contabilidad de red permite decidir sobre qué recursos de red medir y entonces empezar a recoger las métricas relativas a su utilización. Las métricas operan con cuotas que ayudan a asegurar que cada usuario obtenga una *compartición con equidad* de los recursos de la red. Se deben establecer cuotas y en base a ellas penalizar a los usuarios si las superan, por ejemplo denegándoles la utilización del recurso de red en cuestión o cobrándoles una sobretasa adicional. Frecuentemente, las redes de comunicación de datos las utilizan los usuarios para acceder a computadores con algún tipo de base de datos de información. La organización propietaria de la base de datos debe entonces facturar a los usuarios por dichos accesos. Consideremos una organización que mantiene una gran base de datos para consultas on-line. La red establecida para este supuesto se representa en la **figura 1**. Los usuarios deben pagar una cuota de permiso de acceso y la conexión telefónica/telemática (dial-up) a la base de datos. A cada usuario de la red se le permite utilizar la base de datos durante N horas por semana. Esto debe permitir a los usuarios abrir sesión, buscar en la base de datos y por último capturar (transferencia de ficheros 'download') las páginas de información deseadas vía telemática. El tiempo de conexión se utiliza estrictamente para buscar los datos y transferir la información deseada pero no para leer en tiempo real la totalidad de los contenidos de la base de datos. La organización que ofrece el servicio ha estudiado que la métrica de N horas es razonable para la mayoría de los usuarios. Los usuarios que excedan esta cuota deberán pagar una cantidad mensual mayor basada en cuanto tiempo por encima de N horas utilizan. Esto se detecta monitorizando a los usuarios que conectan con sus modems. Los servidores de terminales conectan los modems con la red de área local de la que cuelgan las máquinas-servidores de base de datos. Se puede interrogar a los servidores de terminales cada hora para conocer quién ha conectado y durante cuanto tiempo lo ha hecho. Basándose en esta información se pueden determinar los usuarios que más utilizan la base de datos y calcular el tiempo medio de conexión por usuario y el período de tiempo durante el cual se recibe el volumen mayor de llamadas/conexiones.

Facturar a los usuarios por la utilización que hacen de la red. Normalmente se suelen recoger regularmente los datos de facturación. La mayoría de los dispositivos de red poseen contadores estadísticos que permiten ser sondeados para

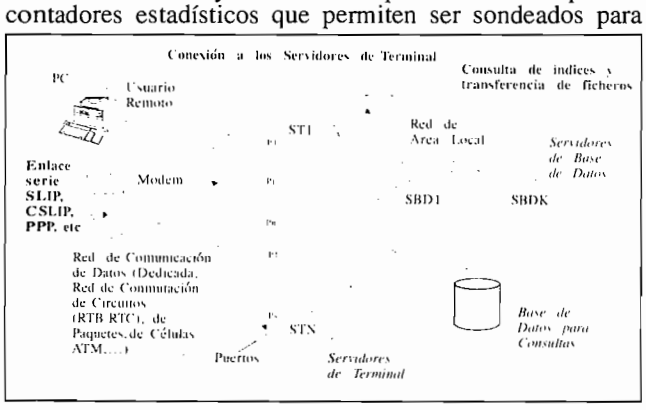


Figura 1: Esquema del acceso a una Base de Datos

pedirles información tanto de forma síncrona como asíncrona y entonces se observan los cambios ocurridos desde el último sondeo. Para ayudar a realizar ésto, el dispositivo de red mantiene una tabla de contabilidad que registra pares de valores (dirección fuente, dirección destino) junto con el número de transacciones, paquetes o bytes que se han intercambiado. A los usuarios se les suele facturar en base a dos criterios: cuota de instalación inicial y cuotas mensuales; o cuotas basadas en la cantidad de recursos de red consumidos.

4. Técnicas de facturación a usuarios de una red

Cuota de instalación inicial y cuotas mensuales. Al usuario se le cobra una cierta cantidad por la instalación ó enganche a la red y luego una cantidad determinada por cada mes de utilización. Empleando este método, la gestión de la contabilidad no es necesaria para la facturación. Aunque éste es el sistema más fácil de implementar no es equitativo ya que un usuario que utiliza la red de forma contínua se le factura la misma cantidad que a un usuario ocasional.

Cuotas basadas en la cantidad de recursos de red consumidos. Este planteamiento es más equitativo ya que se factura a cada usuario en función del grado de utilización de los recursos de red. A veces esta técnica se combina con el cobro de una cuota de instalación inicial (fija) y cuotas mensuales. En Iberpac por ejemplo, se cobra una cuota de alta como usuario de la red que se paga una vez. La tarificación mensual consta de una cuota fija cuyo coste esta en función de la velocidad de transmisión (bps) contratada y una cuota variable que depende del volumen de información transmitida independientemente de la distancia. Para poder implementar esta técnica se requiere recoger a cada usuario estadísticas relativas a la utilización que hacen de la red. Se emplean como **métricas** para determinar la utilización de los recursos de red tres criterios, usados individualmente o de forma combinada: número total de transacciones, número total de paquetes, número total de bytes.

La medida del número total de **bytes o paquetes** se puede basar en la cantidad de información que el usuario envía hacia la red o en el volumen de información que la red envía hacia el usuario. La contabilización del número total de **transacciones** de cada usuario permite a las organizaciones encargadas de la gestión de la contabilidad de red medir en base a otros criterios, como pueden ser el número de 'logins' (aperturas de sesión) a un servidor de computación, el número de conexiones realizadas a un controlador de cluster de terminales, el número de mensajes de Correo Electrónico enviados, el número de sesiones de 'Telnet' y/o 'Login remoto' establecidas, etc.. Un ejemplo de este planteamiento de facturación se representa en la **figura 2**. A pesar de que este planteamiento ofrece la ventaja de ser relativamente fácil de implementar, presenta el inconveniente de que a cada transacción se factura la misma cantidad sin tener en cuenta el tiempo o recursos utilizados. Por tanto, si un usuario realiza una única transacción que envía 800 Mbytes de información, a ese usuario se le factura lo mismo que a otro usuario que tan sólo envía 200 Mbytes de Correo Electrónico. Naturalmente tal estrategia de facturación no es muy equitativa.

Si se mide el número total de **paquetes** la facturación reflejaría la utilización real de la red. Cada vez que un usuario envía o recibe un paquete, la factura aumenta. Este método presenta un inconveniente: la factura para un número dado de paquetes es la misma independientemente de la cantidad de información enviada o recibida. P.ej. un usuario puede enviar paquetes pequeños para soportar tráfico interactivo, el cual, no sobrecarga los dispositivos o enlaces de la red. En cambio otro usuario que

inicia una transferencia de ficheros necesita paquetes de mayor longitud. Si tanto el tráfico interactivo como la transferencia de ficheros precisa 6000 paquetes para realizarse, sus facturaciones coincidirían. Sin embargo, el funcionamiento interno en ambos casos es distinto: la transferencia de ficheros precisa paquetes más largos con lo que utilizará más recursos de red y sin embargo se les facturará lo mismo que al tráfico interactivo, que utiliza paquetes más cortos y por tanto menos recursos.

Los inconvenientes de los dos primeros métodos pueden evitarse facturando los bytes totales utilizados. Con este método al consumidor se le factura en base a la cantidad de recursos de red utilizados. Se plantea aquí una cuestión relativa a si se debe facturar el número total de bytes enviados o recibidos. El facturar el tráfico total en ambas direcciones es redundante en casi todas las configuraciones de red. Existen ventajas evidentes en ambas políticas de actuación. Facturar los bytes enviados a la red de comunicación de datos intuitivamente tiene sentido cuando el usuario envía algo a través de la red, con lo que su facturación se incrementaría. Desafortunadamente en el modelo de red cliente-servidor ahora predominante, esta estructura de facturación presenta algunos inconvenientes importantes. Facturar una salida genera unos gastos a los usuarios que ofrecen sus servicios utilizando sus propios servidores. Supongamos que un usuario WG7 de una red dispone de un servidor de ficheros que contiene datos importantes para un proyecto de investigación: muchos usuarios se conectan a dicho servidor de ficheros y capturan grandes cantidades de información cada día. Los usuarios que la recogen envían paquetes pequeños al servidor pidiendo información (comunicación interactiva). El servidor de ficheros entonces transfiere grandes volúmenes de datos de vuelta al usuario que lo ha pedido. Si la facturación se basa en los bytes que salen del usuario WG7, que ofrece la información, éste recibirá una gran factura que tendrá que repartir entre los usuarios que le han pedido información. Alternativamente, se puede facturar a los usuarios en base a los bytes recibidos de la red de comunicación de datos. Este método elimina el problema de los usuarios que al ofrecer sus servidores pagan por la recepción de información de sus clientes. No existe facturación por el envío de grandes cantidades de datos a la red, sino sólo por lo que reciban. En una configuración de red convencional donde se factura por los bytes recibidos, consideremos una organización central que posee un gran computador de gama alta llamado C1. Los usuarios se comunican con C1 a través de un procesador frontal que conecta con una serie de controladores de cluster con terminales dispersos a través de un ámplio mapa geográfico utilizando enlaces de datos dedicados de bajo ancho de banda. Los paquetes más pequeños se envían desde los terminales a C1. Los mayores volumenes de información se reciben de vuelta de C1 al terminal solicitante.

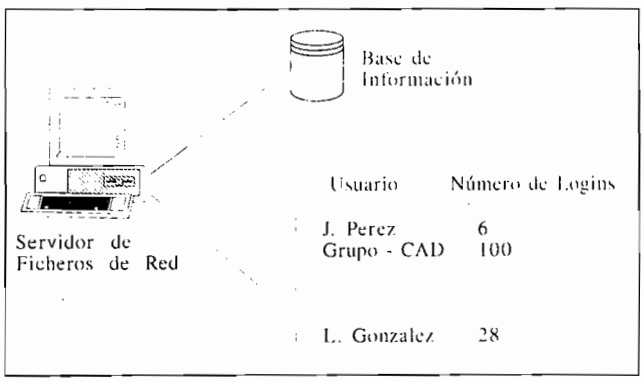


Figura 2: Modelo de facturación en función del número de logins

Los usuarios de muchos entornos de red pueden configurar un computador para que permita acceso a todos los usuarios a ciertos ficheros sin un permiso explícito. En redes IP (Internet Protocol) un método para realizarlo es utilizando 'FTP-(File Transfer Protocol) Anónimo'. En redes Appletalk, el mismo efecto se consigue a través de compartir carpetas. En un servidor de ficheros Novell se puede dejar a los clientes un directorio desprotegido para que depositen o recuperen ficheros libremente. La mayoría de estos servicios se diseñan para que cualquier usuario pueda recoger libremente documentos y aplicaciones. Si se facturan los bytes recibidos de la red, se asegura que cada usuario que ofrezca un servicio no tenga que pagar una facturación importante. Sin embargo, el facturar por los bytes recibidos presenta algunos inconvenientes. En primer lugar, muchos protocolos de red envían reconocimientos (ACKs) desde el receptor al emisor, lo que da lugar a que los usuarios que ofrecen servicios a la red reciban bytes de datos de la red que no han solicitado. Afortunadamente, los paquetes de reconocimiento son normalmente bastante pequeños y estos bytes pueden ignorarse. Los dispositivos de red pueden calcular el número total de reconocimientos. El mecanismo de gestión de la contabilidad de red que se encarga de calcular las facturas puede identificar los usuarios que ofrecen servicios a la red y posiblemente ofrecer un descuento en sus facturas teniendo en cuenta dichos reconocimientos. Otro problema relacionado con la facturación basada en bytes recibidos es que los datos de red no solicitados, tales como Correo Electrónico se añaden a la factura del usuario. Este inconveniente se puede eliminar debido a que muchos usuarios envían y reciben el mismo volumen de Correo Electrónico. Cuando un usuario se encuentra inscrito a una Lista de Correo Electrónico recibe muchos mensajes de correo. En este caso, el usuario se encuentra en la Lista de Correo Electrónico por su propio interés con lo que su factura reflejaría la recepción de estos datos como resultado de un servicio de valor añadido de la red pedido. Aún existe otro problema con este planteamiento de facturación cuando un usuario recibe datos de la red como consecuencia de que la organización encargada de la gestión de red le monitoriza por razones de gestión. Muchos de estos datos se enviarán al usuario regularmente cada cierto período de tiempo dado (las interrogaciones pueden generarse con diversa periodicidad, por ejemplo, una vez al día, una vez cada hora, una vez cada N minutos, etc.). Es posible calcular la cuantía de las interrogaciones que se envían normalmente durante el período de facturación y cuantos bytes los forman. Por ejemplo, si la organización factura mensualmente, fácilmente puede calcular el número medio de interrogaciones enviadas en un mes dado y restar estos bytes a cada factura. En teoría, todos los demás bytes enviados por razones de gestión suceden mientras el gestor de red esta analizando problemas de red, que es un servicio al usuario. Cuando se implementa un esquema de facturación basado en recursos, la gestión de la contabilidad se necesita para poder recoger las estadísticas necesarias. Esto entonces requiere que la organización obtenga la información de los recursos, la procese y produzca las facturas basadas en los recursos consumidos.

5. Jerarquía de mecanismos de gestión de contabilidad de red con diversos grados de prestaciones

5.1. Mecanismos de nivel-1

El mecanismo del nivel de prestaciones más bajo (Nivel-1) permite monitorizar para cualquier métrica el rebasamiento de una cuota de utilización. Los datos de la métrica se almacenan en la base de datos relacional que es parte de la arquitectura

del sistema de gestión de red y la métrica propiamente dicha la configura el gestor de red. Para poder determinar si se han rebasado las cuotas, se utiliza una interrogación SQL con lo que el mecanismo mostrará los resultados de la interrogación. Por ejemplo, si se necesita monitorizar el número de usuarios conectados a un servidor de aplicaciones. Se puede utilizar un mecanismo de gestión de contabilidad muy sencillo para que interrogue al servidor una vez cada cierto tiempo, determine el número de usuarios y coloque estos datos en la BD relacional. A continuación se puede utilizar ese mecanismo para buscar en la base de datos relacional usando esta orden SQL:

SELECT tiempo, número_de_usuarios
FROM estadísticas_del_sistema.

El mecanismo de Nivel-1 muestra el número total de usuarios que cada cierto tiempo utilizan el servidor de aplicaciones. Debido a que los datos de métrica de utilización se guardan en la base de datos relacional, estas estadísticas se pueden visualizar y utilizar de diversas formas, por ejemplo estableciendo métricas y cuotas relativas al número máximo de usuarios registrados en este servidor de aplicaciones. Análogamente, se puede estar interesado en conocer cuando un usuario ha tenido más de k intentos sin éxito de 'login' a través de la red a una base de datos confidencial. Se puede mantener a este mecanismo de gestión de la contabilidad de red monitorizando esta estadística indicándole que interrogue al servidor de la base de datos confidencial cada cierto tiempo para detectar posibles intentos sin éxito de 'login remotos'. Este mecanismo de Nivel-1 almacena esta información en la base de datos relacional. La siguiente interrogación SQL producirá la información de contabilidad necesaria de la tabla de 'logins_incorrectos' de la base de datos relacional:

SELECT usuario, número_de_intentos FROM
logins_incorrectos WHERE número_de_intentos>k

El resultado obtenido es un listado de nombres de usuario seguidos por el número de intentos de 'login' fallidos cometidos, si éste fue mayor que k.

5.2. Mecanismo de nivel-2

Un mecanismo de gestión de la contabilidad de nivel de prestaciones intermedio (Nivel-2) permitirá realizar la facturación de la red. Implementar un proceso de facturación en una red puede ser muy difícil y requerir mucho tiempo. Un mecanismo de Nivel-2 reduce esta carga tomando como entrada la topología de la red y los dominios de facturación y entonces calcula las facturas necesarias a cada usuario. Este tipo de mecanismos necesita datos del sistema de gestión de red y del gestor de red para realizar sus funciones. Obtiene la topología de red de la base de datos del sistema de gestión de red. A continuación el mecanismo debe saber como se divide la topología lógica de la red en dominios de facturación, fase que también requiere entrada de información por parte del gestor de red que delimita una región de facturación en el mapa de red eligiendo un perímetro que rodee el grupo de dispo-sitivos, computadores y enlaces de red bajo análisis. Esto especificará el lugar donde debe sondear el mecanismo de gestión de la contabilidad. Por ejemplo, un grupo a facturar se compone de computadores que residen trás un único disposi-tivo de red. Dicho mecanismo tendrá en cuenta este hecho y sondeará el único dispositivo de red para recoger las estadís-ticas necesarias. El mecanismo podrá determinar donde sondear para recoger información de tarificación. Consideremos una red local formada por un colectivo de computadores interconectados utilizando dos concentradores (o Hubs), por ejemplo de par trenzado (10BaseT). Utilizando el mapa de red se puede seleccionar una región que represente una planta de un edificio donde la totalidad de computadores se conectan a uno de los dos concentradores. El mecanismo de gestión de la contabilidad puede deducir que la mejor estrategia es sondear sólo a uno de los dos concentradores. Sin embargo, suponiendo que se le pide al mecanismo que genere la información de facturación para una región que posee 5 computadores, 3 conectados a un concentrador y los otros 2 conectados al otro. En este caso, el mecanismo determina que el planteamiento más adecuado consiste en interrogar a cada computador de forma individual para poder recoger la información necesaria.

Muchas redes contienen enlaces, lazos y dispositivos redundantes que sólo funcionan cuando otro dispositivo ha fallado. Estas redundancias pueden dificultar al mecanismo a la hora de aislar a qué dispositivo interrogar. Si el mecanismo encuentra alguna dificultad para realizar ésto, preguntaría al gestor de red. Una vez que se ha acotado una región de la red, el mecanismo preguntaría sobre:

- Información relativa a cómo facturar, p.ej. de acuerdo a la entrada en bytes, la salida de paquetes, total de conexiones, etc.
- La frecuencia del sondeo para recoger los datos, por ejemplo semanalmente, cada día, por horas, etc..
- La información de precios para la región considerada, por ejemplo costo por byte, por paquete, etc..

A patir de aquí el proceso de facturación para la región considerada comenzará, es decir, los datos de facturación se recogerán y colocarán en la base de datos relacional. Este proceso se representa en el ordinograma de la **figura 3**.

5.3. Mecanismos de nivel-3

Un mecanismo de gestión de la contabilidad de red de nivel de prestaciones avanzado (Nivel-3) permite predecir las necesidades relativas a recursos de red. Esta capacidad de predicción puede ayudar a establecer métricas y cuotas más eficaces y razonables. Otra característica de este tipo de mecanismos es poder ayudar a los usuarios a predecir sus costos de facturación de red. Los datos de métricas y cuotas pueden ayudar a determinar si los recursos de red son suficientes. Al igual que los mecanismos de nivel-1 de prestaciones utilizando la base de datos relacional pueden producir estadísticas que señalen la frecuencia con que los usuarios han superado las cuotas durante un período de tiempo especificado. Los mecanismos de nivel-3 de prestaciones mejoran la funcionalidad anterior de modo que pueden determinar si una tendencia de la red da lugar a que se alcance una cuota y de este modo alerta a que se actualice el recurso añadiendo más prestaciones (bancos de memoria, potencia y número de microprocesadores, etc.) a dicho recurso, cambiando la cuota o haciendo otra posible acción que se determine como necesaria. Por ejemplo, consideremos una red de comunicación de datos privada dentro de una organización. Los usuarios utilizan una batería de modems para conectar con máquinas de tratamiento de la información para recuperar datos. Parece razonable pensar que a los usuarios se les asignará un límite máximo de tiempo de utilización del modem. Desde que se establece la conexión a través de un modem se dispone de un tiempo límite de N unidades para realizar la llamada/conexión. Al final de las N unidades de tiempo el modem desconecta la llamada. Durante muchos años este límite había sido adecuado en dicha organización y ninguna transacción se había impedido que se complete. Sin embargo, con el tiempo la organización ha evolucionado y ha establecido convenio con otras. Como parte de este convenio los Megabytes de información se transferirán por la noche. Al principio, las transferencias sólo duraban M<N unidades de tiempo pero en la actualidad con el paso del tiempo se necesita que se envíe más información con lo que la transacción supera el límite de tiempo asignado para cada modem. Sin embargo, puesto el

mecanismo de gestión de la contabilidad de red de nivel-3 de prestaciones lo detectó a su debido tiempo, las transferencias pueden continuar sin interrupciones. El mecanismo ha detectado que el tiempo de conexión se acercaba a la cuota e informó para que se modifique urgentemente dicha cuota ya obsoleta. En este caso, la información de gestión de la contabilidad condujo a una decisión de redistribución de la capacidad. Los mecanismos de gestión de la contabilidad de nivel-3 también pueden predecir una facturación de red para los usuarios.

Para poder realizar ésto, el mecanismo opera en dos fases:
- Examina la base de datos relacional para determinar cualquier tendencia en la facturación de un usuario concreto observando los ciclos de facturación anteriores.
- Toma todos los datos disponibles para el ciclo de facturación presente y extrapola al final de un ciclo de facturación concreto

Supongamos que el gestor de un grupo de usuarios pide al mecanismo que realice una predicción de la próxima facturación de red del grupo. Se introducen los datos del grupo en el mecanismo el cual busca la información de facturación en la base de datos de gestión de red y encuentra todos los registros pasados de ese grupo. Supongamos que estos registros muestran un incremento del 6% para cada uno de los ocho últimos ciclos de facturación. El mecanismo a continuación verifica la información en la base de datos para el ciclo de facturación corriente. Extrapolando esta información al final del ciclo el mecanismo encuentra un 7% de incremento probable desde el último ciclo de facturación. Esto se correlaciona con el 6% de incremento del pasado. Como salida el mecanismo produce una facturación para el ciclo de facturación pedido que incluye un 7% de incremento sobre el último ciclo de facturación. Sin embargo, puede que la extrapolación de información no se correlacione con las cuotas de facturación pasadas. Esto puede suceder si el ciclo de facturación corriente ha empezado y los datos de la muestra no reflejan un ciclo de facturación completo o el grupo ha tenido un aumento o disminución de la actividad de la red. En estos casos el mecanismo necesita sacar una predicción basada en los datos históricos y en los derivados del ciclo de facturación corriente.

6. Consideraciones sobre los informes sistetizados

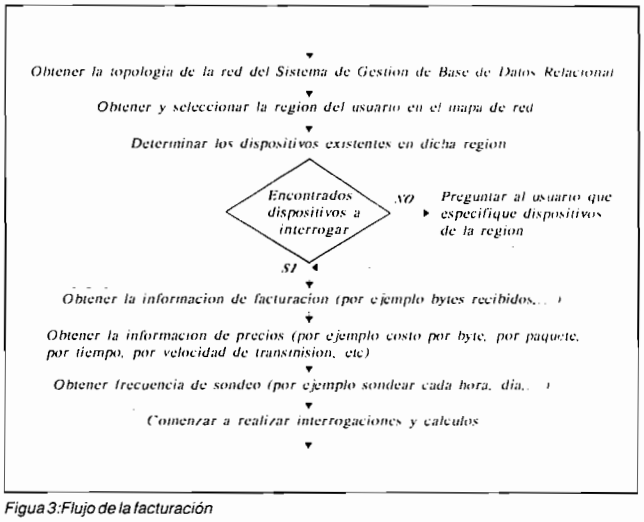
Los informes de la gestión de la contabilidad de red pueden presentarse en forma de mensajes en tiempo real (que pueden especificar el valor de una cierta métrica) o en forma de texto (que pueden ofrecer contabilidad e información de facturación histórica). Algunos de éstos sobre estadísticas de gestión de la contabilidad son resúmenes históricos de métricas, otros pueden predecir tendencias en el empleo futuro de un recurso de red. Se puede usar esta información para planear cuotas realistas para los recursos de la red. Los también importantes informes que proporcionan a cada usuario su facturación de red muestran la información utilizada para calcular la facturación corriente y predecir el precio de la siguiente facturación. Los campos de dichos informes de facturación de utilización de red suelen ser:
- Período de facturación {(día-mes-año inicial),(d-m-a-final)}.
- Número total de bytes recibidos de la red (Mbytes).
- Dispositivos sondeados para determinar el valor del período de facturación (routers, pasarelas, hubs, IWUs, Brouters, ...)
- Precio del Megabyte.
- Facturación total en Ptas.
- Ultimo período de facturación en Ptas.
- Incremento porcentual (XX%).
- Predicción de la facturación del próximo período en Ptas.
- Predicción del incremento porcentual (YY%).

7. Perspectivas

Si se lleva a cabo un análisis riguroso de las diversas plataformas de gestión de red actuales, se puede detectar, en general, una insuficiente potencia para poder gestionar apropiadamente las tasas de transacciones y los tamaños de registros requeridos por los proveedores de servicios de red. Esto explica el hecho de que muchos proveedores de servicios para poder satisfacer los requerimientos de rendimiento necesarios continúan utilizando plataformas propietarias, construidas a medida basadas en grandes máquinas de tratamiento de la información para poder implementar sistemas de facturación de red y de gestión del servicio adecuados. Se observa no obstante hoy en día, la aparición paulatina de productos de gestión de red y aplicaciones de facturación y gestión del servicio de red con capacidades escalables en las plataformas de gestión de red para hacer frente a las diversas necesidades planteables con un enfoque hardware y software avanzado. Este artículo ha abordado una cuestión de vital importancia para un número creciente de organizaciones modernas como es la gestión de la contabilidad de redes mostrando las fases del procedimiento que lo conforma, su impacto e importancia actuales y algunos mecanismos de gestión de contabilidad clasificados y estructurados en orden a su grado/nivel de prestaciones en tres jerarquías: baja (Nivel-1), intermedia (Nivel-2) y elevada (Nivel-3).

8. Bibliografía

AREITIO, J.; AREITIO, A.M. *"Herramientas avanzadas de análisis para redes de ordenadores"*. REE.nº470.pp.64-67. Enero 1994.
SCHWARTZ, M.; *"Telecommunication Networks: Protocols, Modeling and Analysis"*. Addison-Wesley Publishing Company. Reading. Mass. 1987.
GREINER, R.; METES, G. *"Enterprise Networking"*. Digital Press. Digital Equipment Corporation. 1992.
DAIGLE, A.; *"Queuing Theory for Telecommunications"*. Addison-Wesley Publishing Company. Reading Mass. 1990.
FORTIER, P.; DESROCHERS, G. *"Modeling and Analysis of Local Area Networks"*. CRC Press. Boca Raton. Fl. 1990.
RHODES, P.; *"LAN Operations"*. Addison-Wesley Publishing Company. Reading. Mass. 1991.



Figua 3:Flujo de la facturación

Informaticus Mundi

Eliminación de barreras a la integración socio-laboral de discapacitados motrices: soluciones tecnológicas

José Mª Fernández Meroño, Joaquín Roca, Pedro Díaz, Miguel Moreno, José A. Vera
Dpto. de Automática, Electricidad y Electrónica Industrial, Escuela Politécnica Superior de Cartagena Universidad de Murcia
E-mail: jmfm@plc.um.es

Resumen: *se presentan las líneas generales de la soluciones basadas en tecnologías de vanguardia para el desarrollo de un sistema que permita, a través del ordenador, el uso de software estándar en equipos informáticos para discapacitados motrices.*

1. Antecedentes

En la actualidad, las conquistas y avances del denominado *estado del bienestar* obligan a las Administraciones Públicas a actuar sobre los sectores de la sociedad menos favorecidos e integrarlos, en la medida de lo posible, en los entornos socio-laborales que denominamos *normales* de la sociedad.

En este trabajo sólo vamos a ocuparnos de la aportación que la tecnología actual ofrece, con sus continuos avances, en el diseño de equipos que permiten al usuario realizar determinadas labores con plena autonomía. Generalmente, en todos los países, la planificación de esta integración se plantea haciendo concurrir en programas de investigación a grupos de trabajo de Universidades u otras Instituciones de forma que las soluciones se complementen y establezcan normalizaciones que permitan un abaratamiento en los costes de los equipos o sistemas presentados. Por otro lado, las Administraciones dictan leyes de contratación laboral con ventajas fiscales para aquellas empresas que contratan a discapacitados para determinados puestos de trabajo.

Nuestro grupo de I+D, desarrolla un Sistema de Comunicación Integral (SISTCOM), a utilizar por paralíticos cerebrales u otros discapacitados que presenten similares limitaciones en el aparato motriz. Este proyecto está financiado por la Consejería de Industria y Comercio, a través del Instituto de Fomento de la región de Murcia, y por la Fundación ONCE.

En este trabajo, vamos a presentar algunos de los módulos que están siendo ensayados por paralíticos cerebrales tutelados por la Asociación ASTUS, cuya participación en este proyecto nos ha permitido conocer las exigencias de los usuarios y poder así fijar los criterios de diseño.

2. Criterios generales de diseño

- Los equipos se dispondrán lo más discretamente posible evitando espectacularidad y volúmenes innecesarios.
- Las intercomunicaciones se realizarán por R.F., infrarrojos, etc., evitándose las conexiones vía cableado.
- Se evitará las modificaciones en los puestos de trabajo a compartir en las empresas con personas sin discapacidades.
- Aplicaremos criterios de diseño de bajo costo y componentes estándar, con facilidad de adquisición.
- Se intentará universalizar las soluciones, haciéndolas programables y lo suficientemente flexibles para que se adapten a gran número de usuarios.

3. Esquema general del sistema

Este sistema se divide en dos grandes apartados (**figura 1**):
- la **Unidad Móvil de Comunicación (UMC)**
- el **Puesto de Trabajo Multifuncional (PTM)**.

La **UMC** está instalada sobre la propia silla de ruedas, e incorpora el captador adaptado a las necesidades de cada usuario, el sistema de acceso por barrido y un sistema de transmisión por radiofrecuencia. Adicionalmente, y a fin de posibilitar la comunicación verbal, también incorpora un sintetizador de voz, controlable igualmente por técnica de barrido, operando éste en modo conversión texto-palabra.

El **PTM** está constituido por un sistema informático convencional, conectado a una Interfase Controladora del Puesto, que realiza las funciones de EMULADOR DE TECLADO Y RATON a partir de las señales recibidas desde la UMC. Este puesto de trabajo, puede ser utilizado, según el software estándar que en ellos se ejecute, tanto en acciones formativas, educación especial, lúdicas, profesionales, control industrial, etc., incluso artísticas y creativas. Actualmente ya se encuentra con un primer prototipo operativo de Controlador de Puesto, capaz de comunicarse por radio, en una estructura de red tipo LAN, con el ordenador situado en la UMC.

Para las pruebas del PTM se conectó una unidad de transmisión a un PC independiente (PC1), que actuaba como el de la UMC. La ejecución de un software de barrido apropiado en este PC permitió enviar los códigos de operación, vía enlace de radiofrecuencia, a un segundo PC (PC2) que simulaba el PTM (**fig. 2**).

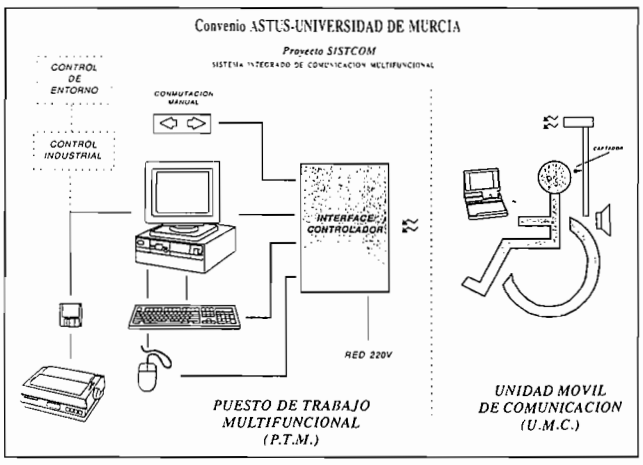


Figura 1: Esquema general del sistema. UMC y PTM

En éste, se recibían las señales transmitidas por radio, que eran pasadas vía RS232C, a un microcontrolador. El software residente de éste decodificaba la información recibida, transformándola en las señales adecuadas para controlar la ejecución del software estándar en el PC2.

El modelo de Controlador de Puesto se estructuró en torno a un DS5000 de Dallas Semiconductor. Este componente es un microcontrolador de 8 bits que es compatible a nivel de software y hardware con la arquitectura del 8051. Está equipado con un puerto serie *full-duplex* con niveles TTL, dos *timers* y otras prestaciones muy interesantes. Las pruebas realizadas con el mismo dieron resultados muy satisfactorios.

Tanto la UMC como el PTM se sontrolan con la señal generada por el discapacitado por medio de un captador o conmutador. Pudiéndose controlar mediante técnicas de barrido la utilización de cualquier sistema manejado por un microcontrolador, y en nuestro caso un PC. Los usuarios de estos sistemas presentan una gran variabilidad en los grados de habilidad para accionar estos captadores. Para el diseño correcto hay que tener en cuenta la idónea ubicación corporal y factores tales como estética, movimientos espásticos, tiempos de respuesta, etc. Es por consiguiente, necesaria la colaboración de un equipo médico que a través de pruebas específicas, nos faciliten los datos que permitan el diseño del captador mas adecuado.

Podemos adelantar que en la actualidad, disponemos de cinco tipos de captadores distintos, siendo nuestra intención es presentar una gama de los mismos que cubran todas las limitaciones de los usuarios, pudiendo seleccionarse como productos estándar de catálogo, a través de una referencia.

4. Técnicas de barrido

Mediante esta técnica, el usuario discapacitado puede acceder a todas las opciones de un sistema informático aunque cuente con una única capacidad residual.

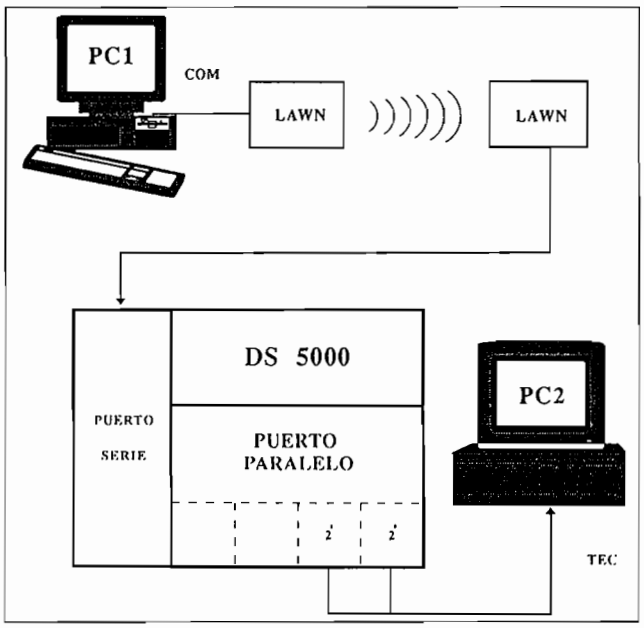


Figura 2:

La pantalla de barrido, consta de un elemento de representación gráfica (monitor, tablero con LED's, etc.) en el que se muestran las distintas opciones que pueden ser elegidas. Cada una de ellas es activada en pantalla secuencialmente, por las señales de un Generador de Barrido guiado por un Reloj de Barrido, que funciona con una frecuencia prefijada, acorde con la velocidad de respuesta del usuario discapacitado (**figura 3**).

Simultáneamente, las señales del generador de barrido son aplicadas al Selector de Barrido. Este dispositivo, que presenta una estructura propia de un multiplexor, conmuta secuencialmente tras cada señal de barrido, la entrada única con una de las distintas salidas existentes (una por opción del menú de barrido). Encontrándose en todo momento alineados ambos dispositivos (pantalla y selector), cualquier señal aplicada a la entrada única desde el captador será dirigida a la salida correspondiente y a la opción seleccionada en la pantalla de barrido.

Su funcionamiento es como sigue: el usuario discapacitado observa el Menú de Selección de Opciones que aparece en la Pantalla de Barrido. Cuando ve que la opción que desea seleccionar ha sido activada por el reloj de barrido, activa el conmutador, el cual produce una señal que aplicada a la entrada única del Selector de Barrido desencadena la ejecución de la opción seleccionada. La velocidad de barrido puede ser regulada de forma que se ajuste a la velocidad de respuesta de cada usuario, e incluso a la evolución de la habilidad del mismo en el manejo del sistema.

En caso de que el usuario posea más de una capacidad residual, el sistema de barrido puede ser optimizado empleando las nuevas capacidades para simplificar el control del mismo.

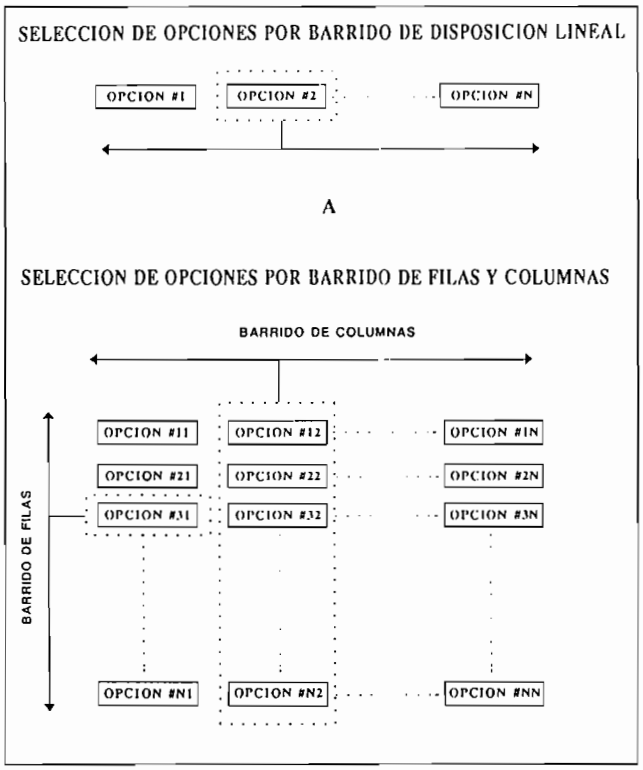


Figura 3

En casos simples y de pocas opciones, el barrido puede adoptar una disposición lineal como se observa en la **parte A** de la **figura 3**, pero es más habitual adoptar la disposición de filas y columnas como se muestra en la **parte B**. Como se observa en ésta, el barrido comienza activando cada columna de la pantalla de barrido secuencialmente (cambio de color, encendido de led's, etc.). Si se aplica un pulso al captador cuando la columna está activa, se iniciará un barrido por filas que permite la selección de la acción deseada por un segundo pulso.

En aplicaciones muy complejas, cuando se precisa controlar un número de opciones muy elevado, se explota una primera pantalla de barrido de zonas, a partir de la cual se accede a una segunda pantalla de barrido de opciones. Cuando el proceso se realiza sobre una pantalla de ordenador, generalmete se usa para esto la técnica de los menús desplegables.

En caso de trabajar sobre una pantalla con punteros de diodos led's se accede a una segunda pantalla paralela. Mediante estas técnicas se desarrollan Teclados Virtuales que facilitan las funciones de acceso.

5. Emuladores

En principio, el uso del conmutador y de los sistemas de barrido permiten sin dificultad el acceso al software específicamente desarrollado o adaptado para discapacitados. Esto resulta satisfactorio en la etapa de enseñanza o aprendizaje, pero a medida que la persona discapacitada se va haciendo más experta o adulta, le será más interesante poder acceder al software profesional estándar comecialmente disponible.

Ante la imposibilidad de adaptar todo el software estándar disponible, se recurrió a la técnica de diseñar un software residente que realizase la función de sustituir totalmente las señales procedentes de teclado y ratón, por otras generadas a partir de conmutadores y por sistemas de barrido.

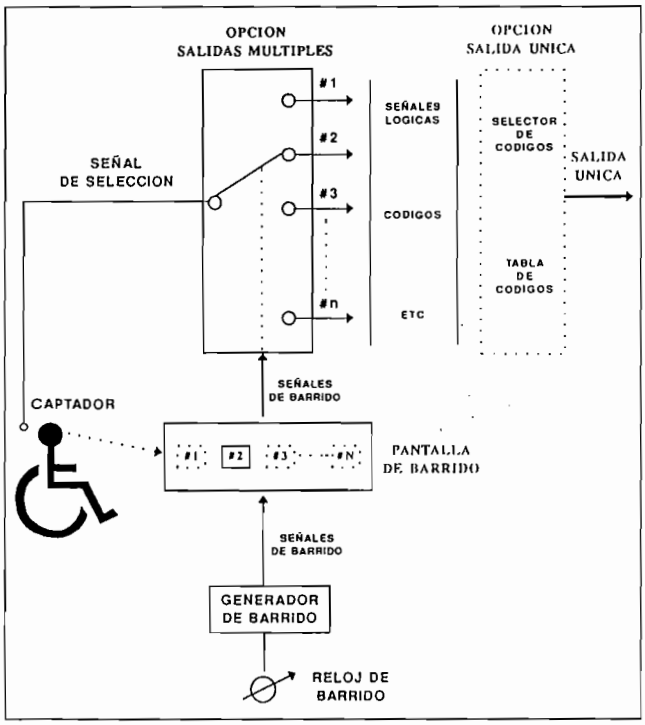


Figura 4

En estas condiciones, para sustituir el teclado en un proceso de textos, por ejemplo, se diseñará un software tranparente que muestre en parte de la pantalla la estructura de un Teclado Virtual, controlado por técnicas de barrido, en el que se puede operar a partir de las señales producidas por un conmutador, mientras que en el resto de la pantalla se ejecuta el software estándar al que se desea acceder.

El concepto de transparencia aquí empleado hace referencia al hecho de que ambos, el software estándar y el teclado virtual, pueden ser ejecutados simultáneamente sin producir interferencias mutuas. El problema surge cuando se intenta utilizar otro software residente, tal como ocurre cuando un nuevo software ocupa o utiliza posiciones de memoria e interrupciones no usadas por versiones anteriores y que fueron utilizadas en su día en la instalación y operación del software transparente.

En la actualidad, se tienden a utilizar Sistemas de Emulación Externos, capaces de generar exactamente las mismas señales que da el periférico que debe ser emulado, como se observa en la **figura 4**, aplicándolas a los conectores en los que normalmente se conectan estos periféricos y dejando que sea el propio sistema operativo del ordenador el que los procese, ignorando si proceden de un teclado o ratón convencional.

De otra parte, el teclado de barrido se implementa sobre una pantalla adicional de LCD, por lo que en el ordenador sólo se carga el software estándar a utilizar (lúdico, formativo, didáctico, etc.) asegurándose así una total compatibilidad.

A este respecto se vienen utilizando equipos de transmisión por infrarrojos o radiofrecuencia (400-900Mhz) y la tendencia actual es la de instalar el captador y el sistema de transmisión sobre la misma silla de ruedas utilizada cotidianamente por el discapacitado.

6. Conclusiones

Las tendencias actuales en el desarrollo de equipamientos, capaces de permitir el acceso al ordenador como herramienta en la Educación Especial e Integración Social y Laboral del minuválido, deben orientarse a posibilitar el uso de software y equipos estándar, mediante la utilización de las herramientas y técnicas descritas.

7. Bibliografía

J.S. COIN; *"Desarrollos en hardware y software para personas discapacitadas"*. Novática V.17, nº 90. 1991.

J. Roca, J.M. Fernández, P. Díaz y otros; *"Sistemas de emulación de periféricos de entrada"*. IX Jornadas de Automática. La Manga del Mar Menor. 1993.

Derecho privado

Isabel Hernando
Experto en la Comisión/DGXIII (Luxemburgo)

Productos multimedia: licencias y derechos de autor [1]

1. Introducción

Con cáracter general, el espacio jurídico genérico al que se circunscribe la obra multimedia, entendida como áquella que dispone y reune en un mismo soporte, varios elementos numerizados: textos, sonidos, imágenes fijas, secuencias animadas de imágenes y otros datos informatizados, cuyo acceso está activado por un programas de ordenador, es el de la Propiedad Intelectual. Este espacio, estimado natural, es aplicable siempre que el criterio de originalidad esté presente y, sin perjuicio de una posible utilización del régimen de la Propiedad Industrial.

Ahora bien, la explotación del producto multimedia, particularizado por la numerización y por su cáracter interactivo, viene condicionada por el 'estatus' jurídico específico seleccionado dentro del sector mencionado. Como consecuencia, la problemática jurídica en el tema que nos ocupa, de identificación de los correlativos derechos de explotación y, de las subsiguientes licencias, discurre en los dos aspectos básicos de identificación del estricto régimen jurídico aplicable al producto multimedia y, del derivado de su elaboración; ambos delineados a continuación.

2. Régimen Jurídico del producto multimedia

Tanto a nivel nacional como comunitario existe una incertidumbre acerca de la regulación propia de la obra multimedia. Esta incertidumbre conlleva la aparición de diversas opciones, que con mayor o menor acierto, vinculan al multimedia a los regímenes jurídicos específicos siguientes: (1) programas de ordenador, (2) obra audiovisual, (3) obra base de datos.

2.1. Obra multimedia y programas de ordenador

La asimilación de la obra multimedia a los programas de ordenador goza de escaso predicamento por su carácter reduccionista al limitar el variado y amplio contenido del multimedia a uno de sus elementos que, aunque imperativamente necesario, no es el exclusivo.

No obstante, esta posible vinculación implica el sometimiento de la obra multimedia a la regulación establecida por la Ley 16/1993, de 23 de diciembre de incorporación al Derecho español de la Directiva 91/250 CEE, de 14-V-1991, sobre protección jurídica de programas de ordenador de la que se desprende su particular régimen de derechos de autor y de licencias [2].

2.2. Obra multimedia y obra audiovisual

La segunda opción de equiparar la obra multimedia a la obra audiovisual, entendida como creación expresada mediante una serie de imágenes asociadas, con o sin sonorización incorporada, destinada esencialmente a ser mostrada a través de aparatos de proyección o cualquier otro medio de comunicación pública de la imagen y del sonido, con independencia de la naturaleza de los soportes materiales de dicha obra [3], conlleva la aplicación a la obra multimedia del régimen jurídico establecido en la LPI de 1987 para la obra audiovisual y, en especial de lo estipulado en sus artículos 86 a 94 y 112 a 115[4]. En particular, el artículo 88 de la LPI dispone que los derechos de reproducción, distribución y comunicación pública de las obras audiovisuales se presumen cedidos en exclusiva a los productores y, el artículo 90 (3) reconoce el derecho de los autores a la remuneración por la proyección, exhibición o transmisión, debidamente autorizadas, de una obra audiovisual por cualquier procedimiento sin exigir pago de un precio de entrada [5].

Con respecto a la tendencia anterior, este sometimiento de la obra multimedia a la audiovisual posee un mayor atractivo entre los sectores implicados [6], aún cuando se pone en evidencia que su configuración lineal y estática se aleja de la específica de la obra multimedia, numerizada e interactiva.

2.3. Obra multimedia y obra Base de Datos

La tercera y última de las alternativas entronca la obra multimedia en el régimen de las Bases de Datos. Esta afinidad encuentra su fundamento en la definición de Bases de Datos, ofrecida por la Propuesta modificada de Directiva de Protección de Bases de Datos, el 23 de junio de 1993, al entenderla como toda colección de datos, obras y demás materiales ordenados, almacenados y a los que puede accederse mediante medios electrónicos, así como el material electrónico necesario para el funcionamiento de la misma, por ejemplo, su diccionario, índice o sistema de consulta o presentación de información. De la definición quedan excluidos los programas utilizados en la realización o el funcionamiento de la base de datos.

Esta asimilación implica que la obra multimedia al igual que la obra Bases de Datos carece en el ordenamiento jurídico español de una regulación específica, siendo de aplicación a las mismas el régimen de transmisiones y de derechos establecidos, con carácter general, en la Ley 22/1987 de 11 de noviembre de Propiedad Intelectual [7].

No obstante, desde una perspectiva de futuro y siempre que la asimilación se mantenga, la citada Propuesta modificada de Directiva Comunitaria instaura un régimen de protección híbrido: (a) protección de la estructura mediante el derecho de autor y (b) protección del contenido a través de un Derecho 'Sui Generis', en el que es de subrayar el sistema de licencias obligatorias implantado[8].

A. Protección por el Derecho de autor

La Propuesta modificada de Directiva circunscribe la protección por el Derecho de autor a la estructura de la Base

de Datos -(obra multimedia)- y, siempre que la selección o disposición de la colección de obras y materiales sea original.

Cumplido el requisito de originalidad, al titular de los derechos, en cuanto a la estructura se le reconoce el derechos exclusivo a realizar o autorizar los siguientes actos:

(1) Actos de reproducción y de transformación.- En estos la Propuesta revisada engloba las siguientes actuaciones:

(a)La reproducción temporal o permanente de la Base de datos por cualquier medio y de cualquier forma, total o parcialmente.
(b)La traducción, adaptación, reordenación y cualquier otra modificación de la base de datos.
(c)La reproducción de los resultados obtenidos con las actividades anteriores.

(2) Actos de distribución. Estos actos comprenden la distribución de la Base de datos entre el público, realizada por cualquier forma, incluyendo su alquiler o sus copias. A continuación, la Propuesta confirma el principio del agotamiento de derechos.

(3) Actos de comunicación.- Entre estos actos se incluye la comunicación, exhibición o funcionamiento de la Base ante el público efectuada de cualquier forma.

Como excepciones a estos derechos, y sin perjuicio de los que puedan subsistir respecto a las obras o materiales contenidos en la Base, al usuario o adquirente legítimo se le reconoce la posibilidad de efectuar los actos prohibidos siempre que sean necesarios para la utilización de la Base. Su especificación puede venir determinada de las siguientes formas:
(a) Por contrato concluído entre el titular y el usuario.
(b) En defecto de contrato, por lo que se estime necesario para acceder al contenido de la Base y para su utilización.

B. Protección por un Derecho 'sui generis'

El derecho 'sui generis' o derecho a impedir las extracciones no autorizadas del material de una Base de Datos se aplica con independencia de la protección de la Base mediante el derecho de autor y, se refiere a la totalidad o a parte de los materiales contenidos en la Base siempre que no estén protegidos mediante derechos de autor o derechos afines.

Este derecho entendido como el derecho a impedir la extracción o la reutilización no autorizadas de la totalidad o de una parte considerable del contenido de una base de datos con fines comerciales, está sujeto a las siguientes limitaciones:

(1) Licencias de extracción obligatorias.- Por razones de equidad y de no discriminación, se reconoce la obligación de conceder un derecho a extraer o reutilizar, en su totalidad o en parte, las obras o materiales de la base de datos con fines comerciales en los supuestos siguientes:

(a) Cuando las obras o materiales contenidos en la Base de datos puesta a disposición del público no puedan crearse, recogerse u obtenerse de otra fuente independiente y, siempre que la extracción o reutilización no supongan un mero ahorro de tiempo, esfuerzo o inversión financiera.

En este supuesto, junto con la solicitud de la licencia se debe presentar una declaración en la que se expongan con claridad los fines comerciales de la Base.

(b) Cuando la Base de Datos ha sido puesta a disposición del público por los siguientes entes:
* Sector público: autoridades, entes u organismos públicos que, hayan sido creados o autorizados por una disposición legal para recoger o divulgar información, o esté entre sus funciones hacerlo.
* Sector privado: empresas o entidades que disfruten de una situación de monopolio como consecuencia de una concesión exclusiva efectuada por un organismo público.

(2) Excepciones a los actos prohibidos.- Según la Propuesta revisada de Directiva, no precisan autorización del titular de los derechos, las extracciones o las utilizaciones de obras y materiales contenidas en la Base de Datos con fines comerciales, realizadas con los siguientes objetivos:

(a) Fines comerciales.- La extracción y la reutilización con una finalidad comercial es lícita siempre que reuna las condiciones siguientes:
* Efectuada por el usuario legítimo.
* Efectuada sobre partes de menor importancia de las obras y materiales contenidos y,
* Efectuada con cita de la fuente.

(b) Con fines privados.- Tambien es lícita la extracción y la reutilización para fines privados siempre que concurran las circunstancias siguientes:
* Efectuada por el usuario legítimo,
* Efectuada sobre partes de menor importancia de las obras y materiales contenidos en la Base.

En ambos casos (a) y (b), el usuario legítimo debe probar que la extracción y la reutilización no causan un perjuicio a los derechos exclusivos de explotación del titular de la base datos y, que no se extralimitan con ellas los objetivos perseguidos.

3. Licencias en la elaboración del producto multimedia

En la creación de una obra multimedia, en general, se precisa la incorporación de obras y materiales preexistentes que, en la mayoría de los supuestos, están protegidos por el derecho de autor.

A nivel de la legislación interna vigente, la calificación de la obra multimedia, como programa de ordenador, obra audiovisual u obra base de datos, es prácticamente irrelevante a estos efectos. Las obras incorporadas están sujetas a régimen establecido en la Ley de Propiedad Intelectual 1987 y disposiciones conexas a los derechos afines [9].

A nivel comunitario, la Propuesta revisada de Directiva de Protección de Bases de Datos trata de la incorporación de obras protegidas en el artículo 5 distinguiendo las actuaciones siguientes:

(1) Exoneración de autorización.- No requiere autorización de los titulares de los derechos de las obras protegidas, siempre que se indique claramente el autor y la fuente de la cita, de

acuerdo con lo dispuesto en el apartado 3 del art. 10 del Convenio de Berna, la incorporación a una base de datos de los siguientes elementos:
(a) Referencias bibliográficas,
(b) Extractos (con excepción de descripciones o resúmenes sustanciales del contenido o de la forma de obras anteriores)
(c) Citas breves.

(2) Sin exoneración de autorización.-Con carácter general, la incorporación a una Base de Datos de obras y materiales protegidos está sometida a la autorización del titular de los derechos de autor o de otro tipo.

A la vista de lo expuesto, el sistema establecido por la Propuesta revisada de Directiva, al remitir a las leyes internas, no aporta una solución al problema de inseguridad legal planteado a la obra multimedia por la multiplicidad de derechos y de titulares.

Ahora bien, siguiendo en el plano comunitario, para superar las reticencias suscitadas a los autores y a los productores de obras multimedia, se estudian posibles soluciones a nivel de licencias (licencias directas y contractuales, licencias obligatorias y/o legales, colectivas, licencias en cooperación-colaboración) y, de remodelación de los derechos de autor.

Con respecto a este último supuesto, y según el Libro verde de derechos de autor y derechos afines [10], los sistemas técnicos de identificación y de protección de obras conllevan, en el ámbito de la Sociedad de la Información, una nueva conformación de los derechos de reproducción, de comunicación al público, de difusión y transmisión numérica y de radiodifusión numérica.

Notas

[1] Publicado en el Libro *Multimedia en España 1995*. Promovido por Fundesco (próxima aparición).
[2] Vease, para este tema, **HERNANDO, I.,** *Contratos Informáticos*, LC, 1995, págs., 75 a 90.
[3] Art. 86, Ley 22/1987, de 11 de noviembre de Propiedad Intelectual.
[4] Vease, igualmente, la difusión de películas cinematográficas y obras audiovisuales recogidas en soporte videográfico, regulada en el RD., 448/1988, de 22 de abril de 1988.
[5] Vease, sobre la remuneración, **HERNANDO, I.**, ob., cit., pág. 421.
[6] Vease, **LEHMAN, B.A.**, *Intellectual Property and the National Information Infrastructure: The Report of the Working Group on Intellectual Property Rights*, Information Infrastructure Task Force, United States, 1995, págs. 25 y ss; VIVANT, M., *L'incidence de l'harmonisation communautaire en matière de droits d'auteur sur le multimedia*, Commission Europénne, DG.XIII.E.1, Luxembourg, 1995, págs, 45 y ss.
[7] Vease, **HERNANDO, I.**, ob. cit., págs., 408 a 431.
[8] Vease, **HERNANDO, I.**, ob., cit., págs 388 y ss.
[9] Vease, **HERNANDO, I.**, ob., cit., págs 409 y ss.
[10] Vease, **COMMISSION DES COMMUNAUTES EUROPENNES**, *Livre Vert, Le droit d'auteur et les droits voisins dans la Société de l'Information*, Bruxelles, 19-VII-1995, COM (95) 382 final.

Un nuevo libro para 'navegar' entre conflictos:

'Contratos informáticos'

Isabel Hernando, responsable y mantenedora de esta sección durante 3 años, es autora de un 'vademecum' de 900 páginas con toda la Legislación y Práctica del Derecho Informático en materia contractual, editado por Carmelo (tno. 943.219318; fax 943.219512). Aunque sobra su presentación para los lectores de Novática (algunos confiesan que abren la revista por el final para empezar a leerla por el artículo de derecho), no puede menos que reproducirse la presentación de la autora que inicia el propio libro: *"Isabel Hernando, Doctora en Derecho por la Universidad de Paris-2 (Pantheon) y Licenciada en derecho por la Universidad de Deusto, completa su especialización en Derecho Informático y en Derecho Industrial y Contratación Internacional en Estados Unidos, en la Universidad de Texas (Dallas) y en Francia, en la Universidad de Montpellier. Autora de varias monografías y múltiples artículos especializados en diversas revistas,* ***siendo especialmente relevante su habitual colaboración en NOVATICA****, ha presentado ponencias tanto a nivel nacional como internacional, habiendo sido nombrada Miembro de Honor del Center for International Legal Studies de Salzburgo (Austria). En la actualidad compagina sus funciones de experto en Derecho, Tecnología, Industria y Telecomunicaciones en la Comisión de la CEE, como miembro del Legal Advisory Board de la DG XIII (Luxemburgo), con su labor como Profesora titular de Derecho Civil en la Universidad del País Vasco, y la dirección de varios Proyectos de Investigación de Derecho e Informática. Como miembro del 'International Bar Association', está adscrita, entre otros, al Comité 'International Computer & Technology Law".*

Al no poder negar el orgullo producido porque tal curriculum mencione destacadamente la colaboración con Novática, no quedan palabras (ni serían creibles en boca de la revista) para ensalzar el contenido de una obra que tendría que encabezar la biblioteca de nuestras empresas y despachos para evitarnos no pocos quebraderos de cabeza.

Para decir que está 'todo', basta citar sus cuatro grandes partes: Protección jurídica de los programas de ordenador, regulación jurídica de los datos (personales, datos informativos, servicios, BD, EDI), Contratación informática, Criminalidad informática.

Para aclarar 'cómo' está ordenado todo y con qué facilidad conceptual, basta recordar que este libro se inscribe en el proyecto 'Derecho e Informática: la Industria del Software' de la Comisión Interministerial de Ciencia y Tecnología (CICYT) española (e Isabel acaba de quedar finalista en el Premio de Investigación Juan Carlos I).

Para valorar si se puede emplear prácticamente, baste una anécdota: es el libro de cabecera de AVETI, la Asociación Valenciana de las Empresas (pequeñas y medias) de Tecnologías de Información, que se ha puesto a normalizar todos sus formatos de contratación como un nuevo tipo de relación abierta, ética y segura en la profesión. Un detalle final (y cada vez menos exótico, a tenor de la LORTAD): el libro reproduce hasta cuestionarios y solicitudes tipo para Seguro de Riesgos Informáticos de un asegurador alemán. Moraleja: el libro se recomienda con sólo verlo **(J.Marcelo)**